60459960

CONTROL DATA

CDC® CYBER 170 COMPUTER SYSTEMS MODELS 815, 825, 835, 845, AND 855

CDC® CYBER 180 COMPUTER SYSTEMS MODELS 810, 830, 835, 840, 845, 850, 855, 860, AND 990

CDC® CYBER 845S, 855S, 840A, 850A, 860A, 990E AND 995E COMPUTER SYSTEMS

GENERAL DESCRIPTION

HARDWARE MAINTENANCE MANUAL

REVISION RECORD

REVISION	DESCRIPTION
A (07-30-82)	Manual released.
B (02-10-84)	Manual revised; includes Engineering Change Order 44861. This edition obsoletes all previous editions. Because changes are extensive, revision bars and dots are not used and all pages reflect the current revision letter.
C (10-31-85)	Manual revised; includes Engineering Change Order 47463. Front cover, 3-4, 3-17, 7-6, 7-20, 7-22, 7-24, 7-34, 9-3, 9-5, and 9-14 are revised. Index-1 and Index-2 are added.
D (06-13-86)	Manual revised; includes Engineering Change Order 47774. Front Cover is revised.
E (12-08-86)	Manual revised; includes Engineering Change Order 48291. Front Cover through 6 and 7-7 are revised.

Publication No.
60459960

REVISION LETTERS I, O, Q, S, X AND Z ARE NOT USED.

Address comments concerning this manual to:

Control Data
Technical Publications
4201 North Lexington Avenue
St. Paul, Minnesota 55126-6198

or use Comment Sheet in the back of this manual.

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual, are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

PAGE	REV
Front Cover	-
Title Page	-
2	E
3/4	E
5/6	E
7	B
8	C
9	C
1-1	B
2-1	B
2-2	B
2-3	B
2-4	B
2-5	B
2-6	B
2-7	B
2-8	B
2-9	B
3-1	B
3-2	B
3-3	B
3-4	C
3-5	B
3-6	B
3-7	B
3-8	B
3-9	B
3-10	B
3-11	B
3-12	B
3-13	B
3-14	B
3-15	B
3-16	B
3-17	C
3-18	B
3-19	B
3-20	B
3-21	B
3-22	B
4-1	B
4-2	B
4-3	B
4-4	B
4-5	B
4-6	B
4-7	B
4-8	B
4-9	B
5-1	B
5-2	B
5-3	B
5-4	B
5-5	B
5-6	B
5-7	B
6-1	B
6-2	B
6-3	B
6-4	B
6-5	B
6-6	B
7-1	B
7-2	B
7-3	B
7-4	B
7-5	B
7-6	C
7-7	E
7-8	B
7-9	B
7-10	B
7-11	B
7-12	B
7-13	B
7-14	B
7-15	B
7-16	B
7-17	B
7-18	B
7-19	B
7-20	C
7-21	B
7-22	C
7-23	B
7-24	C
7-25	B
7-26	B
7-27	B
7-28	B
7-29	B
7-30	B
7-31	B
7-32	B
7-33	B
7-34	C
7-35	B
7-36	B
7-37	B
7-38	B
8-1	B
8-2	B
8-3	B
8-4	B
8-5	B
8-6	B
8-7	B
8-8	B
8-9	B
9-1	B
9-2	B
9-3	C
9-4	B
9-5	C
9-6	B
9-7	B
9-8	B
9-9	B
9-10	B
9-11	B
9-12	B
9-13	B
9-14	C
9-15	B
9-16	B
9-17	B
10-1	B
10-2	B
10-3	B
10-4	B
10-5	B
10-6	B
10-7	B
11-1	B
A-1	B
A-2	B
A-3	B
A-4	B
A-5	B
A-6	B
Index-1	C
Index-2	C
Comment Sheet	E
Back Cover	-

PREFACE

This manual contains information for the CDC® CYBER 170, Models 815, 825, 835, 845, and 855, the CYBER 180, Models 810, 830, 835, 840, 845, 850, 855, 860, and 990, and the CYBER 845S, 855S, 840A, 850A, 860A, 990E, and 995E Computer Systems.

The manual provides a model-independent overview of the Virtual State System and its security/protection and interrupt features relative to the computer system's hardware.

AUDIENCE

This manual is for use by customer, marketing, training, programming, and engineering services personnel who operate, program, and maintain the computer systems.

RELATED PUBLICATIONS

The following manuals contain related information:

Control Data Publication	Publication Number
CYBER 170 Computer Systems Models 815, 825, 835, 845, and 855 and CYBER 180 Computer Systems Models 810, 830, 835, 840, 845, 850, 855, and 990, and CYBER 990E and 995E Computer Systems Hardware Reference Manual, Virtual State, Volume II, Instruction Descriptions, Programming Information	60458890
CYBER 845S and 855S Computer Systems Hardware Reference Manual, Virtual State, Volume II, Instruction Descriptions, Programming Information	60463410
CYBER 840A, 850A, and 860A Computer Systems Hardware Reference Manual, Virtual State, Volume II, Instruction Descriptions, Programming Information	60463580

Publication ordering information and latest revision levels are available from the Literature Distribution and Services catalog, publication number 90310500.

CONTENTS

APPENDIX

INDEX

FIGURES

This section will be supplied later.

Virtual State contains three main areas:

- Virtual memory mechanism.
- Interrupt system.
- Call/return mechanism.

A complete understanding of each area is necessary to fully comprehend Virtual State.

GENERAL DESCRIPTION

This section deals with the basic concepts of the virtual memory mechanism. The primary purpose of the mechanism is to provide a solution to the security problem. Each executing task of the operating system performs in a unique address space. The address space is divided into a number of segments, each of which can be 2 billion bytes long. The segment forms the basis of the security and protection mechanisms.

To understand memory management, it is necessary to understand the difference between segments and pages. A segment is a unit of virtual memory management. It has attributes such as length and access privileges peculiar to the protection scheme. The page is a unit of real memory management. Pages do not have attributes. Pages are present in the hardware to assist the software in managing the very large real memories that can be supported by Virtual State. The page size is a variable, set during system initialization and constant from one deadstart to the next.

Virtual memory addresses are the only addresses available to software. Virtual State processors do not have a real memory address mode. The only places real memory addresses are used are in the hardware tables used in address translation. The address translation mechanism is discussed in the following pages. To minimize address translation time, since the hardware depends upon good locality of reference, all code and data being used at one time should be collected in virtual memory.

ADDRESS TRANSLATION

The fundamental address available to a programmer is the process virtual address (PVA). To the user, this appears as a segment number and a byte offset within the segment (figure 2-1). It also includes a ring number that is part of the protection mechanism discussed later.

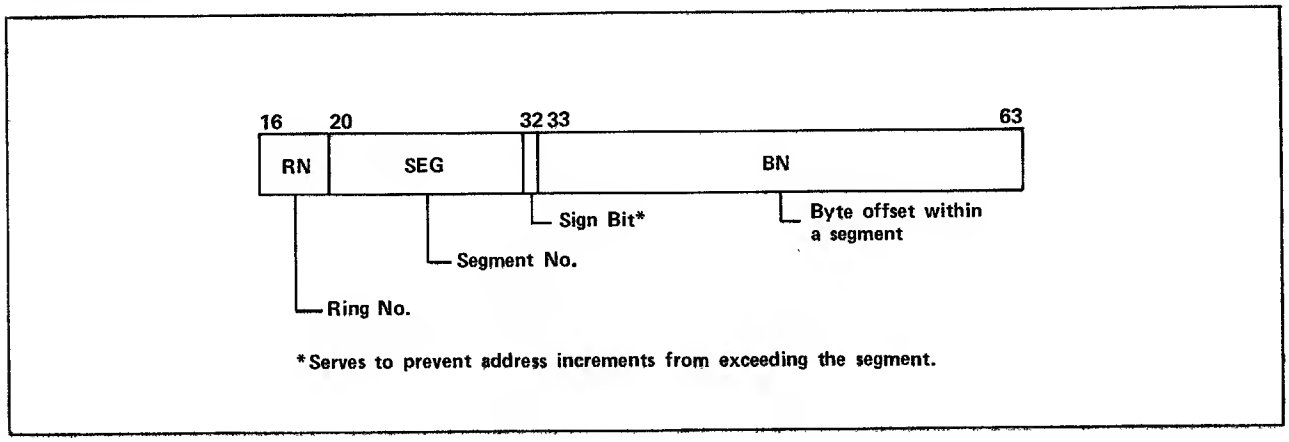


Figure 2-1. Process Virtual Address

A segment is an entity. Address flow is from the beginning of a segment to the end. An address does not flow from one segment to the next segment, and it does not wrap around a given segment. When an address of 31 of more bits is created, an address specification error (ASE) is detected and program execution is interrupted.

The segment number is a 12-bit field used as an index into a process segment table. The operating system creates a process segment table for each active process (task)† in the system. Segment numbers are assigned sequentially from zero.

Segment descriptor table entries (SDEs) are 64 bits long and contain information relating to the privileges and protection of that segment. SDEs also contain active segment identifiers (ASIDs). Each ASID is a unique 16-bit identifier created and used by the operating system to identify each active segment within the system.

Address translation takes place in two steps. The first step uses the process segment descriptor table (SDT) to translate a PVA to a system virtual address (SVA). SVAs form the basis for code sharing, and processor cache memories are organized on SVAs, not real memory addresses.

†The terms process and task are synonomous. A task is the unit of execution of the Virtual State operating system. A process is the hardware term for a task.

In the first step of the translation, the hardware takes the ASID from the entry in the SDT, pointed to by the SEG field of the PVA. It then catenates this with the BN field of the PVA to form the SVA (figure 2-2).

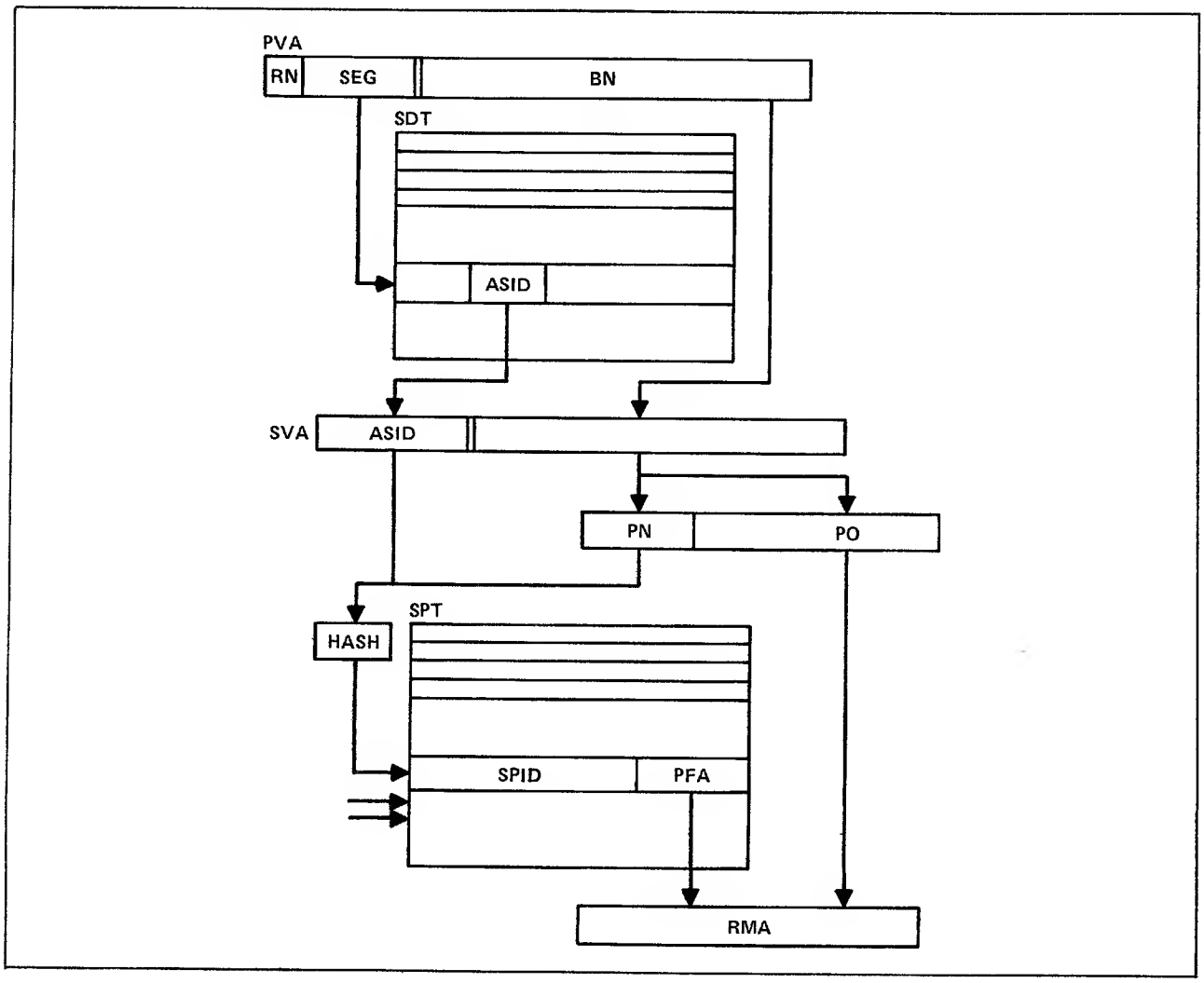


Figure 2-2. Address Translation

The processor views the BN as both a page number (PN) and a byte offset within that page, also called a page offset (PO). Figure 2-3 illustrates the formation of PN and PO for a 4096-byte page. To determine where or if the page resides in real memory, another access to the system page table (SPT) is made. Since many more pages exist in virtual memory than in real memory, a hashing algorithm is used to compute the page table index. On Virtual State, the page number and the ASID are hashed via an exclusive OR. The page table index is used to select a candidate entry from the SPT. The table is then searched forward linearly, either until a valid page with the desired entry is found or until 32 entries have been searched. If the search terminates without a hit, the page is assumed not to be in central memory, and a page fault is indicated.

Once a hit is made, the page frame address (PFA) in the page table is catenated with the page offset from the SVA to form a 32-bit real memory address (RMA).

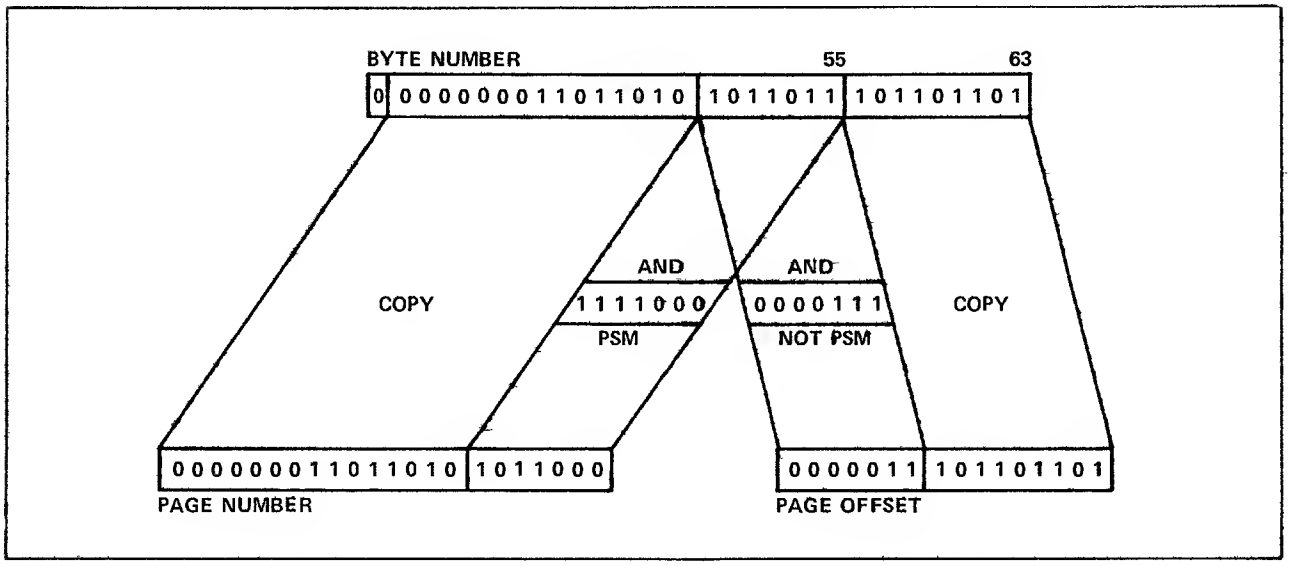


Figure 2-3. Formation of Page Number and Page Offset (for a 4096-Byte Page)

PAGE SIZE

The page size is determined by the value of the page size mask (PSM). The PSM is a 7-bit register that expresses the page size in multiples of 512 bytes. The page size is given by:

$$\text{Page size} = 2^9 \times 2^{7-(+/PSM)}$$

where the PSM is a solid mask extending from left to right. A PSM of 0 indicates the largest page size (64KB). The term (+/PSM) expresses the summation of the 1 bits (pop count) in the PSM.

HASHING ALGORITHMS

The hashing algorithm takes an exclusive OR of the low-order 16 bits of the page number and the ASID. If the page number is only 15 bits long, then a 0 is catenated to the left end to make it 16 bits. Since the resulting hash must be as random as possible, the most random low-order bits of one quantity should be exclusively ORed with the most random high-order bits of another. The low-order 16 bits of the PN are the most random part of that quantity, and the same will be true of the ASID, if it is assigned sequentially starting from zero. This does not allow appropriate randomness in the ASID. To achieve appropriate randomness, the operating system may assign ASIDs from zero up and invert the bits. The first 16 ASIDs then become:

ASID	Hexadecimal	Binary
1	0000	0000 0000 0000 0000
2	8000	1000 0000 0000 0000
3	4000	0100 0000 0000 0000
4	C000	1100 0000 0000 0000
5	2000	0010 0000 0000 0000
6	A000	1010 0000 0000 0000
7	6000	0110 0000 0000 0000
8	E000	1110 0000 0000 0000
9	1000	0001 0000 0000 0000
10	9000	1001 0000 0000 0000
11	5000	0101 0000 0000 0000
12	D000	1101 0000 0000 0000
13	3000	0011 0000 0000 0000
14	B000	1011 0000 0000 0000
15	7000	0111 0000 0000 0000
16	F000	1111 0000 0000 0000

The operating system may also use a pseudorandom number generator for ASID assignment, although the randomness in the high-order bits of the ASID is not guaranteed. The pseudorandom technique is inferior to the hashing algorithm.

The result of the hash is ANDed to the page table length (PTL), and four 0's are catenated to form the page table index (figure 2-4).

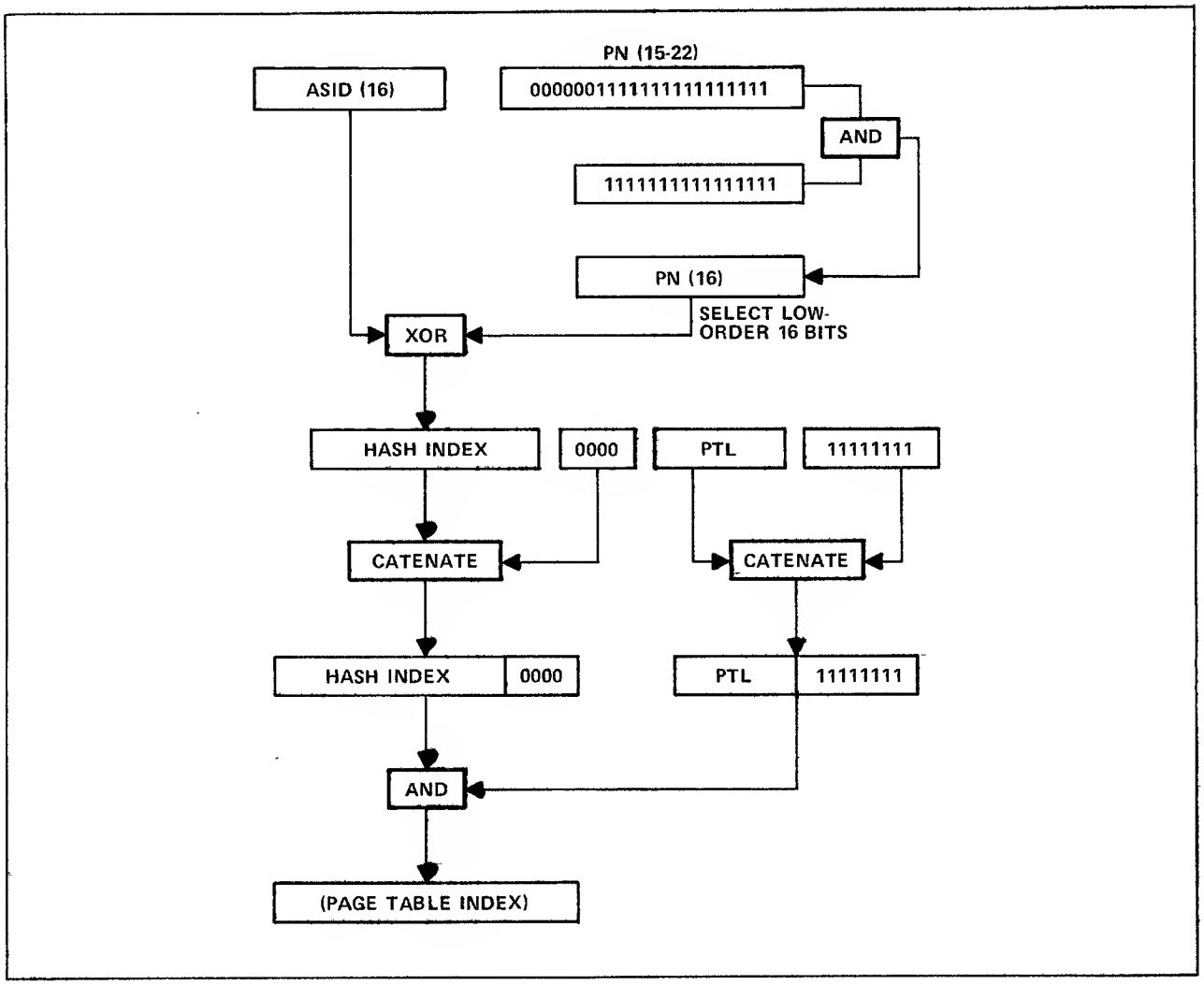
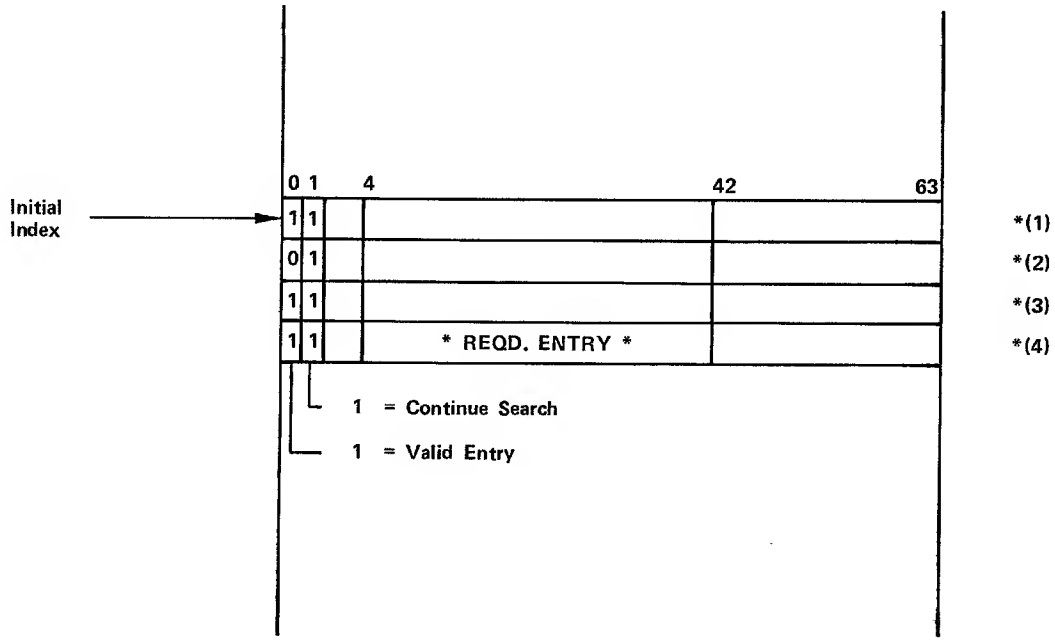


Figure 2-4. Hashing Algorithm

PAGE TABLE SEARCH

Since the hashing algorithm is a many-to-one mapping, two different pages may hash to the same page table entry (PTE). To find the correct entry, the ASID and page number portion of the SVA is compared with the system page identification held in the PTE. If they are not equal, a linear search controlled by the VC field (bits 0 and 1), is initiated. The search continues either until 32 entries have been searched and the correct entry is found, or until an end-of-search condition is indicated by bit 1 (figure 2-5).



*(1) The first entry accessed is valid (bit 0=1) but has the wrong segment page identifier (SPID). Since bit 1=1, the next entry is checked.

*(2) The second entry is invalid, but search can continue.

*(3) The third entry is the same as the first.

*(4) The fourth entry matches the required SPID. The sequence terminates at this point. The continue bit (bit 1) does not necessarily indicate an end of search, since other multiple entries with this hash or an adjacent hash may be present.

Figure 2-5. Page Table Search Example

The algorithm for setting the continue-search bit is self-evident. When an entry is invalidated, its continue bit is checked. If it is set, no additional action is necessary, since it is part of a chain to an entry further down in the page table. If it is zero, the table may be searched backwards to clear out possible continue bits for the now invalid entry. If the previous entry had its continue bit clear, the process terminates, since there is no chain to investigate. If the previous entry had its continue bit set, it is cleared and a check is made to determine whether more continue bits can be cleared. The condition for further clearing is either an ASID of 0 (a null entry by software convention) or an ASID of nonzero that hashes directly to this entry, in which case the continue chain being cleared could have started higher up (figure 2-6). A special system instruction (load page table index) has been defined to aid in this process. The instruction is described in volume II of the Virtual State hardware reference manual (refer to the preface for more information).

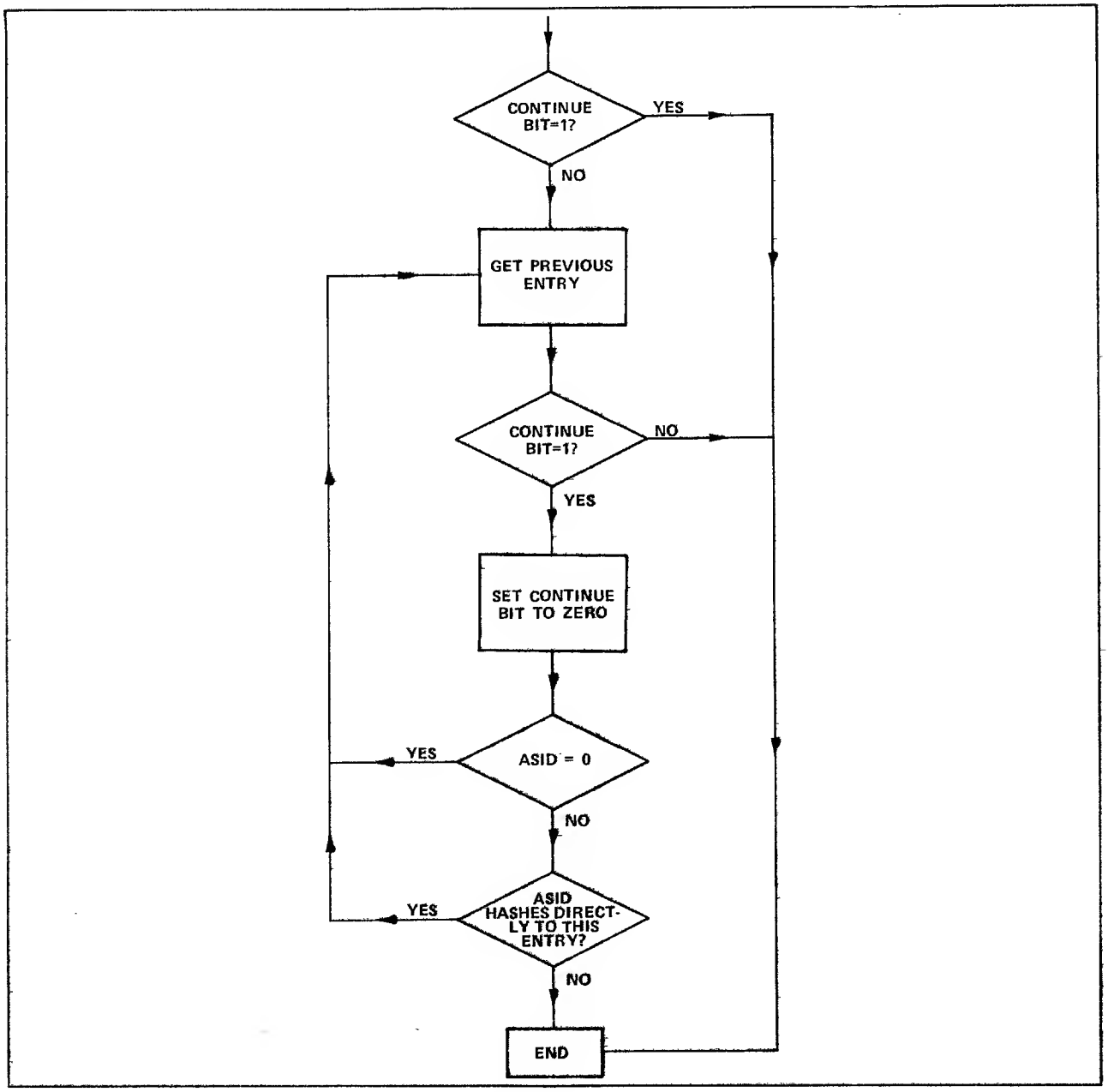


Figure 2-6. Page Table Search Flowchart

In summary, the hardware uses the SDT to translate a PVA into an SVA. It then uses the SPT to translate the SVA into an RMA. The SDT and the SPT are hardware tables constructed and managed by software. The translation of a PVA into an RMA can be time consuming, especially if every reference requires at least two additional memory references. A number of hardware buffers are used to eliminate this overhead and to minimize the time necessary for translation.

To accommodate coincident hash indexes with the minimal search, the operating system assigns from two to four times as many entries in the page table as there are pages in real memory. Since the general environment contains two processors, care must be taken when changing page table entries, and the special interlock instructions must be used for this purpose. The instructions are described in volume II of the Virtual State hardware reference manual (refer to the preface for more information).

The only mode of operation of the hardware is a virtual address mode. No instructions deal directly with real memory addresses. The hardware has been designed with dynamic paging in mind. Pages are brought into memory on demand, and page table entries are purged based on a least recently used (LRU) algorithm. This algorithm is the responsibility of the operating system. However, two flags are kept in the PTE to help in the process. The flags are kept in the UM field (bits 2 and 3) and have the following meanings.

- Whenever a page is used (read, written, or executed), the hardware sets bit 2 in the PTE.
- Whenever a page is modified (written), the hardware sets bit 3 in the PTE.

Combinations of bits 2 and 3 have the following meanings.

Value	Meaning
00	Unused and unmodified (new page)
01	Unused but modified
10	Used but not modified
11	Used and modified

Pages are chosen as candidates for purging based on the value of the UM field and their LRU status. Since any modified page must be written to mass storage when it is purged, modified pages will typically have a higher resistance to purging. The status unused-but-modified can arise from software algorithms. The UM bits are never cleared by hardware. They are cleared by software when pages are purged, and are also cleared to force updates to the LRU status of all pages. In this latter mechanism, it is expected that the operating system will periodically zero all used bits in the page table. This effectively resets the LRU status. Ensuing activity automatically updates this status.

Although the hardware has been designed with dynamic paging in mind, it is not a prerequisite. When running in a pure CYBER 170 State, static paging is used. The entire CYBER 170 State environment is assigned to a single Virtual State segment that operates in Virtual State job mode. Pages in this segment and in real memory have a one-to-one correspondence, and once initialized, the page table does not change. CYBER 170 State operates in a virtual memory segment that has a size corresponding to the amount of real memory in the system. If the ASID is set to HEX FFFF and SEG set to 0, a pseudo RMA mode exists within the hardware. This is a pseudo mode, since the hardware still goes through the address translation mechanism. However, there are no page faults.

Security is particularly important in computer systems. Virtual State has been designed to meet the most stringent security requirements of the industry. The degree of security achieved by a Virtual State system is controlled by the software exploitation of the hardware features. The hardware provides facilities that detect breaches of security, or attempted breaches of security, but the software really controls the desired level of security. If a rigid set of conventions is not followed, loopholes may exist and may be detected by ingenious users. The software and hardware interaction is similar to other controls that must be followed if a system is to be totally secure. These controls embrace the installation management, the operators, the administration, and all other parts of an organization. The computer is only a small part of this whole.

The responsibilities of the organization are not discussed here. Instead, the discussion is confined to the hardware and software facilities provided by Virtual State systems. It is divided into two major areas: the first deals with the software facilities and their interfaces to the end user, and the second deals with the hardware facilities and their use by the software.

SOFTWARE FACILITIES

A basic objective of NOS/VE is to provide efficient and safe services to many users simultaneously and asynchronously. Levels of provided service range from the complete isolation of users from each other to controlled sharing between cooperating users. To allow these service levels, the system has adopted a general access control strategy or security model. The model serves as the conceptual basis for the detailed implementation of all the access control mechanisms in the system.

The access control strategy is based on a conceptual access control matrix. Rows of the matrix represent all possible users of the system. In the access control matrix these are called subjects. Columns of the matrix represent all possible system resources that can be accessed by a subject. In the access control matrix these are called objects, such as subjects, files, and equipment. Each element in the matrix identified by a subject-object pair contains the valid kinds of access or access rights that the subject has to that particular object. Figure 3-1 illustrates access control.

	SUBJECT A	SUBJECT B	FILE C	FILE D	TAPE DRIVE E
SUBJECT A		ADMIN	OWNER R,W		OWNER USE
SUBJECT B			R	OWNER R,X	

Figure 3-1. Access Control Example

In the example in figure 3-1, subject A is the administrator of subject B and the owner of file C and tape drive E. Subject A can read and write file C and use tape drive E. Subject B can read file C and owns and can read and execute file D. Every access a subject makes to an object is validated via the access control matrix. The access is permitted only if the corresponding access right is in the appropriate element in the access control matrix.

The operating system cannot maintain a physical matrix that is consulted on every access. A variety of features of the system architecture interact to implement the conceptual access control matrix. Some of the major features of the NOS/VE implementation of the access control architecture are:

- User identification and validation
  - A user must be known before gaining access to the system.
  - The resources a user can use are a function of user controls, project controls, and the current state of the system.
  - An attribute of every user is the lowest ring number of execution.
  - Modification to the user validation information may only be performed by the system, account, and project administrators who control the user's installation.
- File system
  - All files in the system, local or permanent, are owned by a single user.
  - Access to permanent files by any user other than the owner is regulated by an access control list associated with each file. The access control list contains the names and access rights of all users permitted to access the file.
  - All files, local and permanent, have one or more ring brackets associated with them. The ring brackets are used as qualifiers to file access.

    If a file is readable, then it possesses a read bracket that defines the rings in which it can be read.

    If a file is writable, then it possesses a write bracket that defines the rings in which it can be written.

    If a file is executable, then it possesses an execute bracket that defines the rings in which it can execute, and a call bracket that defines the rings from which it can be called.

    The ring brackets associated with a file are specified by the owner of the file. However, the file system does not allow any user to specify any ring bracket of higher privilege than the ring in which the user is executing.

- All files, local or permanent, possess certain attributes that describe the contents of the file. The ring brackets associated with reading, writing, executing, and calling are all file attributes. Whether a given user has read, write, or execute permission to another user's permanent file is determined by the access control list of the file. The combination of these two factors allows rings to be a unit of protection that is recognizable to the entire system. This is useful to the operating system since it attempts to discriminate between system code and nonsystem code running in a user job, regardless of the user on whose behalf the job is executing. Since the user's installation administrators control the assignment of ring numbers in the validation files, the user controls the extent and connotation of his or her installation ring use. If an installation chooses to associate different rings with different security classifications, it can do so. If it wishes to run all users at a single ring, the only use of rings is to protect the operating system from users' programs.

- Segment management

  - For a file accessed through the system virtual memory mechanism, the file protection attributes maintained by the file system are used by segment management to build the SDEs. The CPU address translation logic uses the SDEs when referencing the segment. The attributes maintained by the file system software are continuously enforced by the hardware when the file is being referenced.

  - The loader accesses all object libraries through the segment level access facility of the file and memory management systems. It also uses segment management to create the transient segments used for the data areas of the executing program. The loader is responsible for creating these segments with the correct protection attributes to assure proper execution and protection of the program.

An example illustrates how these mechanisms interact to effect access control in NOS/VE. Consider two users, Tom and Sue. Tom has been validated by the installation to execute in ring 9. Sue has been validated to run in ring 11. Tom develops an application and stores it in the permanent file catalog. Since Tom was executing in ring 9 when he cataloged his application, it has an execute bracket of (9,9) by default. In order to allow Sue to use his application, Tom must set the call bracket of his application permanent file to 11 and give Sue execute permission in the access control list of the file. Since Tom is the owner of his application, he is the only user permitted to set its call bracket and place entries in its access control list.

In order to use Tom's application, Sue must first attach the file for execute access. This will succeed because Tom has placed Sue in the access control list of the file and has specified execute access. Sue then executes a program that uses Tom's application. Sue's program is loaded in ring 11 and Tom's program is loaded in ring 9. Because Tom's program has a call bracket that extends to ring 11, Sue's program can call Tom's and use the service it provides.

HARDWARE FACILITIES

One of the primary design accomplishments of Virtual State systems is to improve the overall system reliability. In the past, the operating system has been a limiting factor. Operating system software is large, on the order of one million lines of code, and is error prone. An error in other systems often causes the systems to crash, interrupting normal service. Since it is unlikely that such a vast quantity of code can be generated error free, other solutions were sought. In Virtual State, the solution is to give each user his or her own copy of the operating system. If a particular copy of the system fails, it causes nothing more serious than a single job to abort. With this approach, the operating system and the user's code become an entity, and facilities must be provided to separate and protect operating system modules from user modules and from each other. This is the primary reason for the Virtual State security system. A second major objective is to provide controlled access to all code and data. To this end, users are protected from each other, and can be protected from the system.

The key to this protection mechanism is the Virtual State virtual memory mechanism. The virtual memory segment is the basic protected element. Before the attributes of this segment are discussed, however, it is necessary to understand how segments are arranged in virtual memory for utilization by a user.

VIRTUAL MEMORY USER ADDRESS SPACE

Address space is the set of addresses known to an executing process. In CYBER 170 State, this is the set of real memory addresses embraced by reference address (RA) and field length (FL). In Virtual State, it is the set of virtual memory addresses specified by the entries in the SDT. Each process executing in a Virtual State system has a unique SDT. The address and length of this table are specified by process state registers held in the exchange package. The exchange package is used to define the environment of the exchange interval for each task. Each entry in the SDT consists of a full word and describes all of the attributes of one segment. Just as CYBER 170 State users are constrained to an address space by the RA/FL mechanism, so are Virtual State users constrained to an address space by the SDT. This is the basic element of protection. The discussion of the protection mechanisms that follows describes the protection offered within a virtual memory address space.

The following two paragraphs discuss what happens when a user attempts to access code or data segments not in his or her address space. The only addresses known to the user are process virtual addresses (PVAs) (figure 3-2). Each PVA has three components: a ring number (RN), a segment number (SEG), and a byte number (BN).

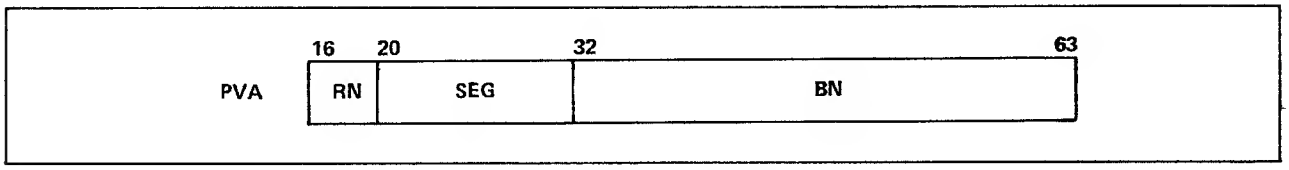


Figure 3-2. Process Virtual Address (PVA)

The ring number is discussed later in this section. The segment number is assigned by the operating system (segment manager) in ascending sequential order starting with zero. The segment numbers are the only names by which the user knows his or her segments. They act as an index into the process segment table that contains entries only for those segments to which the user has access rights. The byte number denotes a byte offset within a segment. The only way a user can attempt to access a segment not contained within his or her address space is by specifying a segment number greater than any assigned by the operating system for that process. However, when an attempt is made to reference that segment, an exchange interrupt results, since the segment number is greater than the segment table length (STL). The STL is set by the operating system and held in a process state register that can be read but not written. The register is set by an exchange jump. The hardware performs this basic test for every reference made to memory.

Whenever an exchange jump occurs, a switch of address spaces occurs. The operating system monitor runs in its own address space. This is not true of the bulk of the operating system. Services such as those offered by record manager reside within the user's address space. Additional protection mechanisms, described later, come into play to protect these parts of the system from the user and vice versa. All of this happens in virtual memory. Conceptually, operating system segments that reside in the user address space exist as multiple copies in virtual memory. To optimize the use of real memory, the operating system typically keeps a single copy of the code in real memory. The copy is shared by several users. The individual users are unaware of this since they are only aware of what happens in virtual memory. There is no possible breach of security here since each user is totally unaware of the existence of other users.

SEGMENT ATTRIBUTES

Once a user has been confined to an address space, the segment becomes the basis of the security mechanism. Each segment has a set of attributes recorded in the segment descriptor table entry (SDE) in the SDT. These attributes are its global system name, its access attributes, its rings, and its lock. Since these are recorded in the SDE, and the SDE is unique to a given process, it is possible for a segment to be shared by more than one process, yet have different attributes for each process. The format of an SDE is shown in figure 3-3.

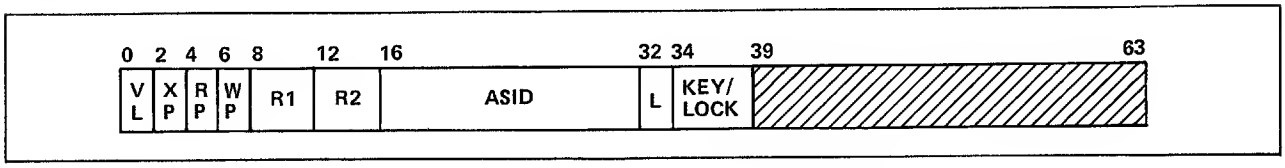


Figure 3-3. Segment Descriptor Table Entry (SDE)

A segment also has a length associated with it that is not kept in the SDE.

The first four fields determine the access privileges for the segment. Values for these fields are:

Field	Value	Description
VL	00	Invalid entry. A segment no longer used by a process does not have to have its SDE removed from the SDT, but must have it invalidated. If segments are viewed as files, their entries are invalidated when the files (segments) are closed and purged.
	01	Reserved.
	10	Regular segment. This denotes an active segment for the executing process.
	11	Cache by-pass segment. Certain tables and interlock words must be kept in a cache by-pass segment. An example is the exchange package. The exchange jump mechanism works from a real memory address. Therefore, data in cache memory, which is addressed via a system virtual address (SVA), does not get updated.

Field	Value	Description
XP	00	Nonexecutable segment. This data segment normally has either read or write access.
	01	Nonprivileged executable segment. Some instructions can only be executed if they reside in a segment having the attributes of local or global privilege. As a result, certain operations are restricted for use by the operating system and cannot be invoked accidentally by a user executing garbage, for example, literals.
	10	Local-privileged executable segment. Code contained in segments with the local-privileged executable attribute may execute all unprivileged instructions and all instructions restricted to local privilege. In particular, trap handlers have at least local privilege, since the trap-enable flip-flop and the trap-enable delay flip-flop can only be set by a system copy instruction, and these flags can only be written in local-privileged mode.
	11	Global-privileged executable segment. Code contained in segments having global privilege may execute all instructions except those restricted to monitor mode. This includes those instructions restricted to local-privileged mode, which is a subset of global privilege.
RP	00	Nonreadable segment. Such a segment has either write privilege or execute privilege. An execute-privilege segment normally has only that attribute. However, literals may be stored in the segment and read from the segment with load instructions provided for that purpose. The load instructions always load from an address relative to P, the program address counter. In this case, the read access is implicitly equated to the execute access of the segment.
	01	Read under control of the key/lock mechanism. For a segment controlled by a lock, this control may selectively apply to either the read or write privilege of the segment. The code indicates that reads are under key/lock control.
	10	Read not under the control of key/lock. This is a normal read privilege assigned to the segment.
	11	Binding section. Read not under the control of key/lock. Binding sections, which contain pointers to external procedures and data, always have read privilege, and are never subject to key/lock control.
WP	00	Nonwritable segment. Typically, all executable segments are nonwritable, because Virtual State code is usually organized into pure procedures. A user could generate a code segment from assembly language that modified code. However, this code segment would not be sharable with other users.
	01	Write controlled by the key/lock mechanism. This is the write counterpart of RP=01 and indicates that the segment is writable but only if the key/lock access is correct.
	10	Write not under control of key/lock. This is a normal, writable data segment.
	11	Reserved.

The XP, RP, and WP are the first level of protection offered within a user address space. Figure 3-4 illustrates segment protection within an address space.

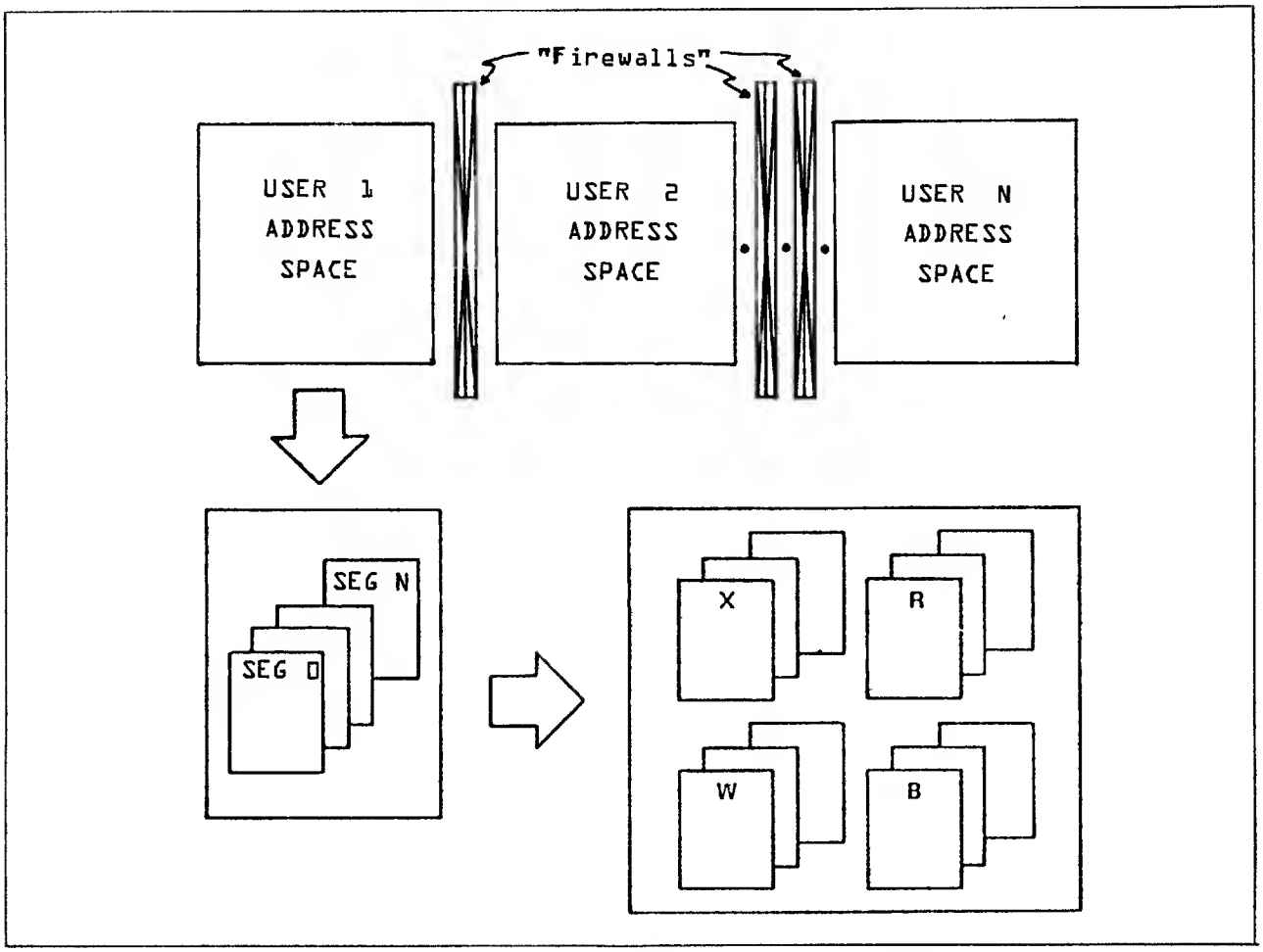


Figure 3-4. Segment Protection within an Address Space

Users are absolutely confined to their address space. Within this space, code and data are organized into segments, and the segments are assigned various privileges. Most of the operating system and entire subsystems are expected to exist in the user address space. These constraints are necessary to ensure that appropriate security is maintained at all times. Even privileged portions of the operating system, such as trap handlers with local privilege, can reside in the same address space as a user who is executing completely unchecked, and therefore very unreliable, code.

This level of protection guarantees that code segments are not arbitrarily overwritten by users, leading to unpredictable results. It also guarantees that read-only data segments are not destroyed, either willfully or accidentally. However, this level of protection is insufficient by itself. For example, users could read and write segments of the operating system, or users with inadequate security clearances could gain access to private data segments. To accommodate these aspects of security, additional mechanisms are provided.

RINGS OF PROTECTION

To provide a separation between code and data segments, and prevent unauthorized access, each address space is organized into a series of rings of protection. A maximum of 15 such rings is permitted in an address space. They may be regarded as separate machine states having differing privileges. The rings are organized hierarchically: the lower the ring number the higher the privilege (ring 1 has the highest privilege). In general, code residing in ring n can read and write segments in ring n and higher numbered rings. In addition, code in ring n can call procedures in ring n and lower numbered rings, although such calling is carefully controlled. The ring protection mechanism is controlled by the R1 and R2 fields in the SDE, the R3 field in the code base pointer (CBP), and the ring number carried in all PVAs, particularly those held in the A registers and P registers. Code and data segments need not reside in a single ring and may exist in several rings. When this occurs, the segment is said to reside in a ring bracket. The extent of this ring bracket is defined by the R1 and R2 fields of the SDE.

Four ring brackets are associated with every segment. These ring brackets are for read, write, execute, and call. The first three of the ring brackets are segment attributes and are described by the SDE. The fourth, the call bracket, is associated with the segment by the operating system with no loss in generality. In practice, the operating system does not only associate these ring brackets with segments but associates them with every file in the system, either local or permanent, whether or not a given file is a segment. Descriptions of the checking performed by the hardware for each type of access follow.

Execute Access

A segment that resides in several rings has its execute bracket described by the R1 and R2 fields of the SDE. Thus if:

$$SDE.R1 \leq P.RN \leq SDE.R2$$

then the segment is a member of the execute bracket. If control is transferred to the segment from within the execute bracket, the ring number in the P register (P.RN) is unchanged. If control is transferred from outside the ring bracket (via an inter-ring call), P.RN is always set to SDE.R2. Calls, if permitted, can only be made inward. The hardware validates execute access only once when a segment is entered. That calls can only be made inward (or to the same ring) appears confusing, but the reason is quite straightforward. Care must always be taken when crossing domains of protection to ensure that no security violation occurs. This is particularly true when crossing from one domain to another with higher privilege. If an outward call is permitted, then its counterpart, an inward return, must also be permitted. However, a return is an unsolicited GO TO, which implies a crossing of domains of protection without the necessary control. It is really an inward return that must be prevented.

Call Access

Two main checks are exercised by the hardware when a call is made. The first ensures that the call is an inward call.

$$PVA.RN \geq SDE.R1$$

where PVA.RN is the ring number of the PVA containing the entry point of the called procedure, and SDE.R1 is the lower range of the ring bracket of the called procedure.

The second check ensures that the caller has adequate privilege to call the callee:

$$PVA.RN \leq CBP.R3$$

where CBP.R3 is the CBP gate ring number.

The callee can restrict entry to the procedure so that he or she can only be called from certain rings. One more case is of interest, where a routine is called on behalf of another caller. This happens when a caller legitimately calls on a more privileged procedure, but then requests that the callee in turn call a third procedure, to which the callee has access but the caller does not. Via a high-level language, this is constructed very simply by a pointer to procedure. To prevent this form of unauthorized call, the hardware performs an additional check.

$$Aj.RN \leq CBP.R3$$

where Aj.RN is the ring number of the pointer used to access the binding section containing the relevant CBP. This A register will not have more privilege than the code requesting the call. It is the requesting code that must reside within the callee's call ring bracket. In practice, since P.RN will always be less than or equal to Aj.RN, the hardware only has to perform the latter test.

Read Access

An executing procedure may read a segment providing the following is true.

$$PVA.RN \leq SDE.R2$$

where PVA.RN is the ring number of the pointer (held in an A register) used to access virtual memory, and SDE.R2 is the outermost ring number for the segment being accessed.

Thus, a procedure may read a segment from a ring of equal or lower privilege than its own. For the read access to be successful, of course, the segment must have the read attribute associated with it.

Write Access

An executing procedure may write a segment providing the following is true.

$$PVA.RN \leq SDE.R1$$

where PVA.RN is the ring number of the pointer (held in the A register) used to access virtual memory, and SDE.R1 is the innermost ring number of the ring bracket for the segment being accessed.

These four major ring brackets are shown in figure 3-5.

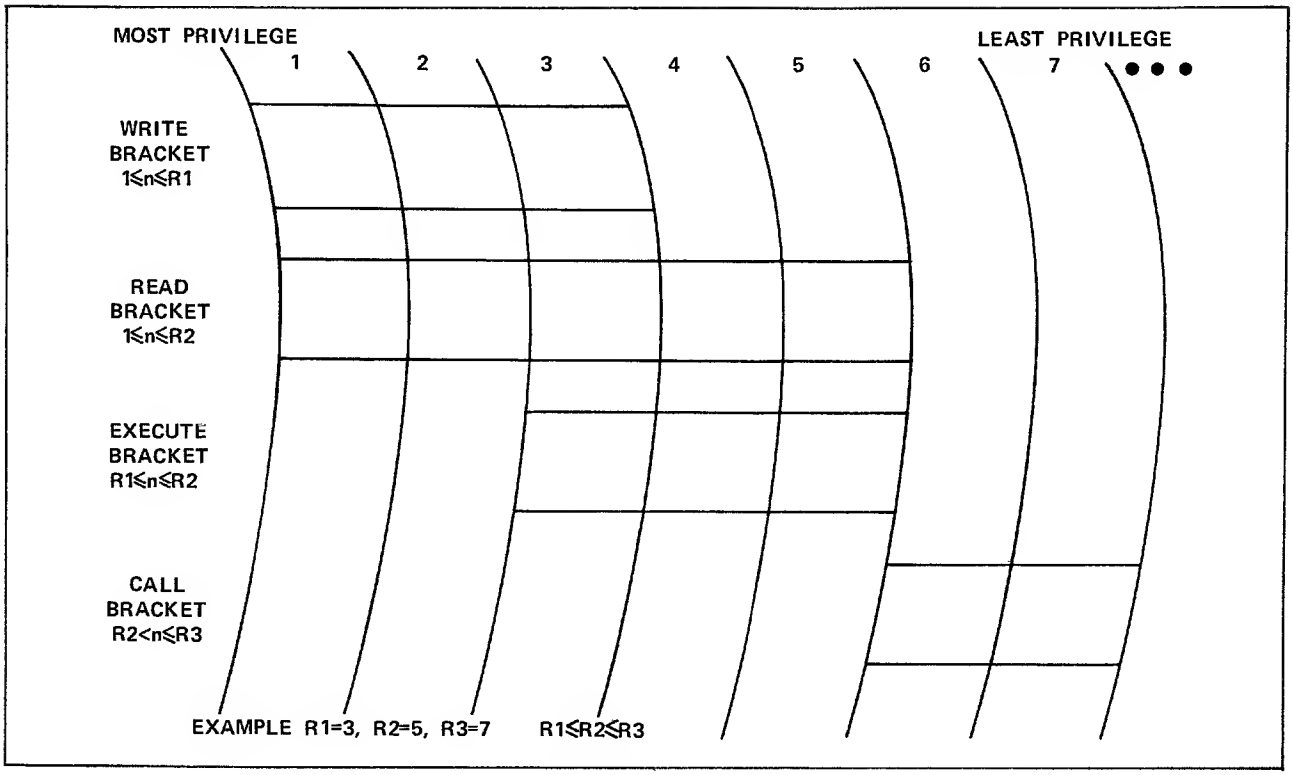


Figure 3-5. Ring Brackets

In this example the R1, R2, and R3 parameters, which define the ring brackets for a particular segment, have been set as follows:

R1 = 3
R2 = 5
R3 = 7

The segment can be written from another segment, if that other segment resides in ring 3 or in a lower numbered ring (ring 1 or 2).

The segment can be read from another segment, providing the other segment resides in ring 5 or in a lower numbered ring (rings 1 to 4).

The segment can execute in rings 3, 4, or 5. If the segment is called from ring 3, it will execute in ring 3; if it is called from ring 5, it will execute in ring 5; and so on. If it is called from ring 6 (or a ring numbered greater than 6), then it will execute in ring 5, the least privileged of rings 3, 4, and 5.

The segment can be called from either rings 6 or 7. Segments can always be called from other segments in the same ring that they are in. Consequently, the segment can be called from rings 3 through 7. It cannot be called from any rings greater than 7, which are outside the call bracket. Neither can it be called from a ring number less than 3, since this would constitute an outward call. This situation is discussed in the call/return section of this manual. Call/return is the primary mechanism for crossing protection boundaries within an address space.

Figure 3-6 shows how ring brackets are used.

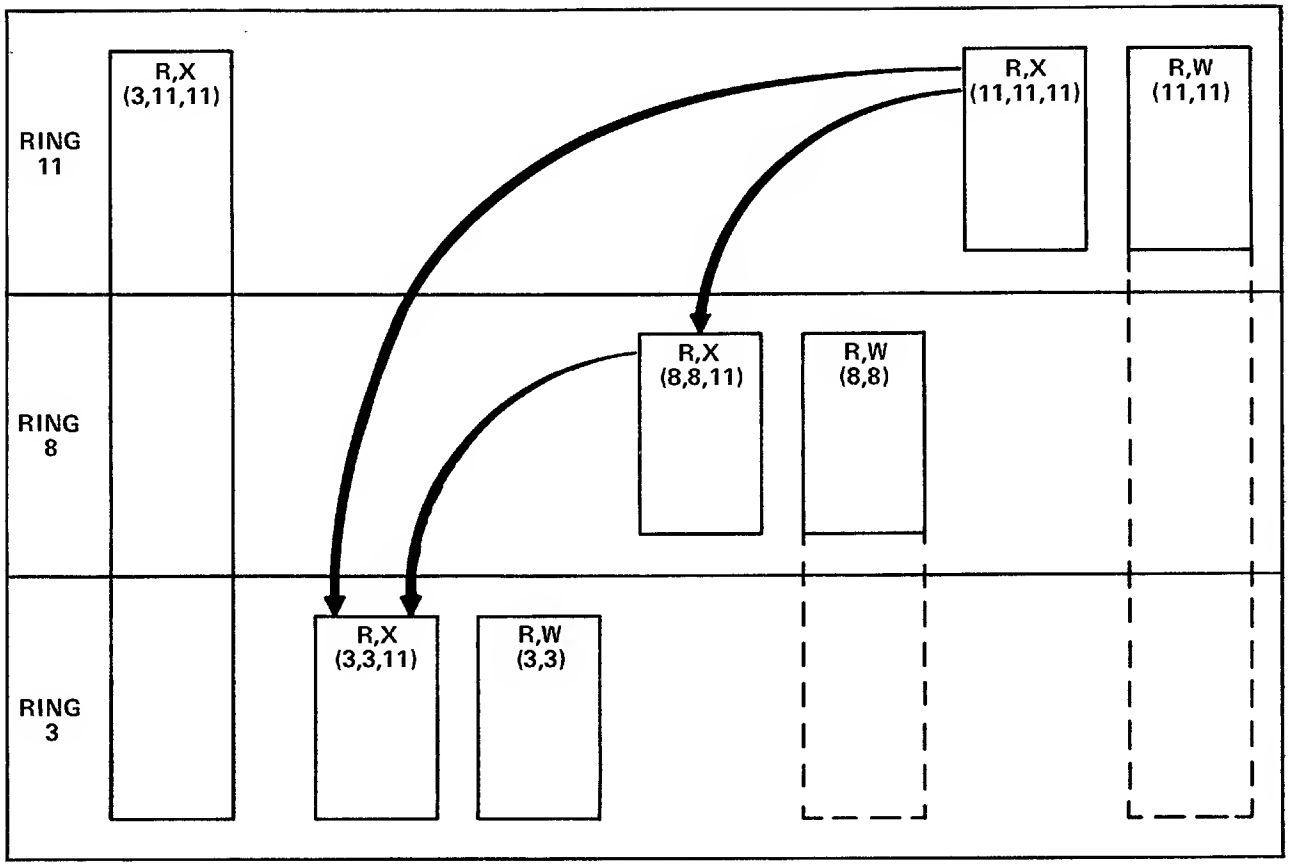


Figure 3-6. Example of Ring Bracket Use

In this example, the extremities of the ring brackets for each segment are indicated by the numbers in parentheses (R1,R2,R3). The procedure in ring 11 can call on the procedure in ring 8, since the call bracket for this latter procedure has been set to 11. The procedure in ring 8 can read and write into the data segment belonging to the procedure in ring 11, since the segment has read/write access and its R1 and R2 fields have both been set to 11. In this way the procedure in ring 11 can pass parameters to and receive results from the procedure in ring 8. Likewise, the procedure in ring 3 can be called from either the procedure in ring 8 or the procedure in ring 11, because the ring 3 procedure has its call bracket equal to ring 11. The logical extensions of the data segments in each ring are indicated by dotted lines in the diagram. Notice that there are no extensions to the execute segments, since the R1 and R2 fields restrict execution of the segment to a particular ring. Thus, the procedure in ring 11 can only be executed in ring 11, the procedure in ring 8 can only be executed in ring 8, and the procedure in ring 3 can only be executed in ring 3.

On the left side of the figure there is an execute segment having R1 and R2 fields of 3 and 11, respectively. Consequently, this procedure can be executed in any ring from 3 through 11, inclusive. Such a procedure might be a trap handler or the FORTRAN math library. A user executing in ring 11 can call on square root, for example, which would then execute in

ring 11. Similarly, a procedure in ring 8 using square root would have it execute in ring 8. In other words, the square root procedure always executes with the privilege of the caller. This is required, since it is acting on behalf of the caller.

The segment can only be read, written, or executed if it has the appropriate access permission associated with the segment. For a segment to be written from another segment, it must contain write permission, and the other segment must reside in a ring from which the first segment can be written.

Within a single address space, therefore, rings of protection provide a mechanism for protecting sensitive code and data. Two cases are of particular interest.

- The first case deals with the need to know. A procedure should have access only to those procedures and data segments necessary to do its task. Remember that the ring mechanism is hierarchical, for the lower the ring number, the higher the privilege. A higher clearance (lower ring number) allows access to more documents, but fewer individuals (segments) are granted such clearance.

- The second case is concerned with degrees of potential damage. The segments of a system can be effectively segregated into two or more rings, according to the damage that may be caused if these segments are misused. The segments whose misuse is likely to cause the greatest damage are given lower ring numbers. By means of this segregation, the bulk of the operating system can reside within the user's address space and yet be protected from the vagaries of undebugged user code. If part of the operating system does fail, the damage can be contained and cause nothing worse than the user job to abort. Rings are therefore used extensively by the operating system for damage control, and also made available for the user to create hierarchical security structure. Figure 3-7 illustrates the resulting user address space.

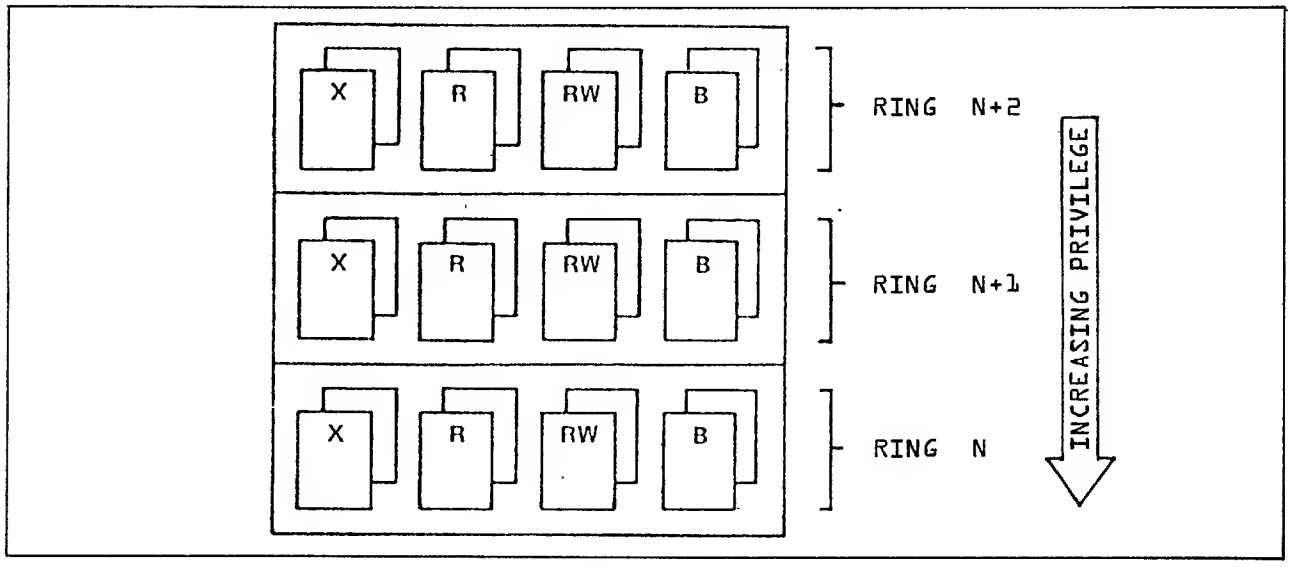


Figure 3-7. Ring Protection within an Address Space

The address space, which is the basic unit of protection, now has two additional protective mechanisms. The first restricts the type of access to a segment, and the second limits the region from which a segment can be accessed.

RING NUMBERS IN POINTERS

The only addresses that programmers deal with are process virtual addresses (PVAs). Figure 3-8 illustrates a PVA. A set of 16 address registers (A registers) exists in the hardware to hold these addresses during instruction execution.

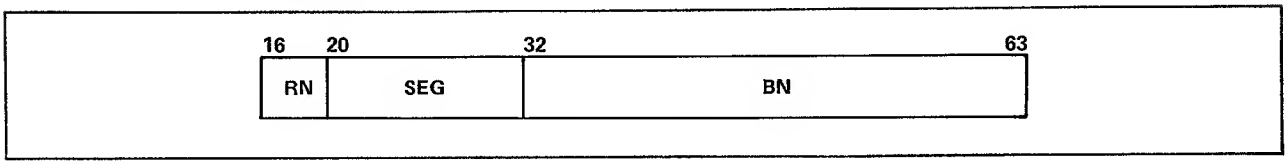


Figure 3-8. Process Virtual Address (PVA)

The SEG and BN fields designate the addressed segment and the byte offset within that segment, respectively. The RN field is the ring number associated with the access being made. This ring number is most important to the ring security in the system, since it is common for a procedure to perform work on behalf of another, less privileged procedure. It is important that the more privileged procedure does not act with greater authority than has been assigned to the caller. As a result, whenever an A register is loaded, either explicitly (via a load or copy instruction) or implicitly (via a return or pop instruction), the hardware places the ring number with least privilege into the register. A comparison is made between the ring number of an A register used for the load, the ring number of the loaded pointer, and the ring number of the R1 field of the SDE associated with the A register used to load the pointer (refer to figure 3-9). The largest of these three ring numbers is entered into the destination A register.

When an A register is loaded via a copy instruction from an X register, a comparison is made between the ring number of the pointer held in the X register and the ring number held in the P register. The larger of the two values is used. Since this cannot be as rigorous a test as that used for loading A registers, care must be exercised in its use. For example, if a procedure calls on a second procedure in a more privileged ring, and a pointer or pointers are passed via loading an X register and copying the X register to an A register, the callee may end up acting on behalf of the caller, with more privilege than the caller is allowed. When this happens, the callee, the more privileged procedure, is at fault. The more privileged procedure must maintain the security of the system down through its own level. Without this fundamental software convention, the hardware cannot maintain system integrity.

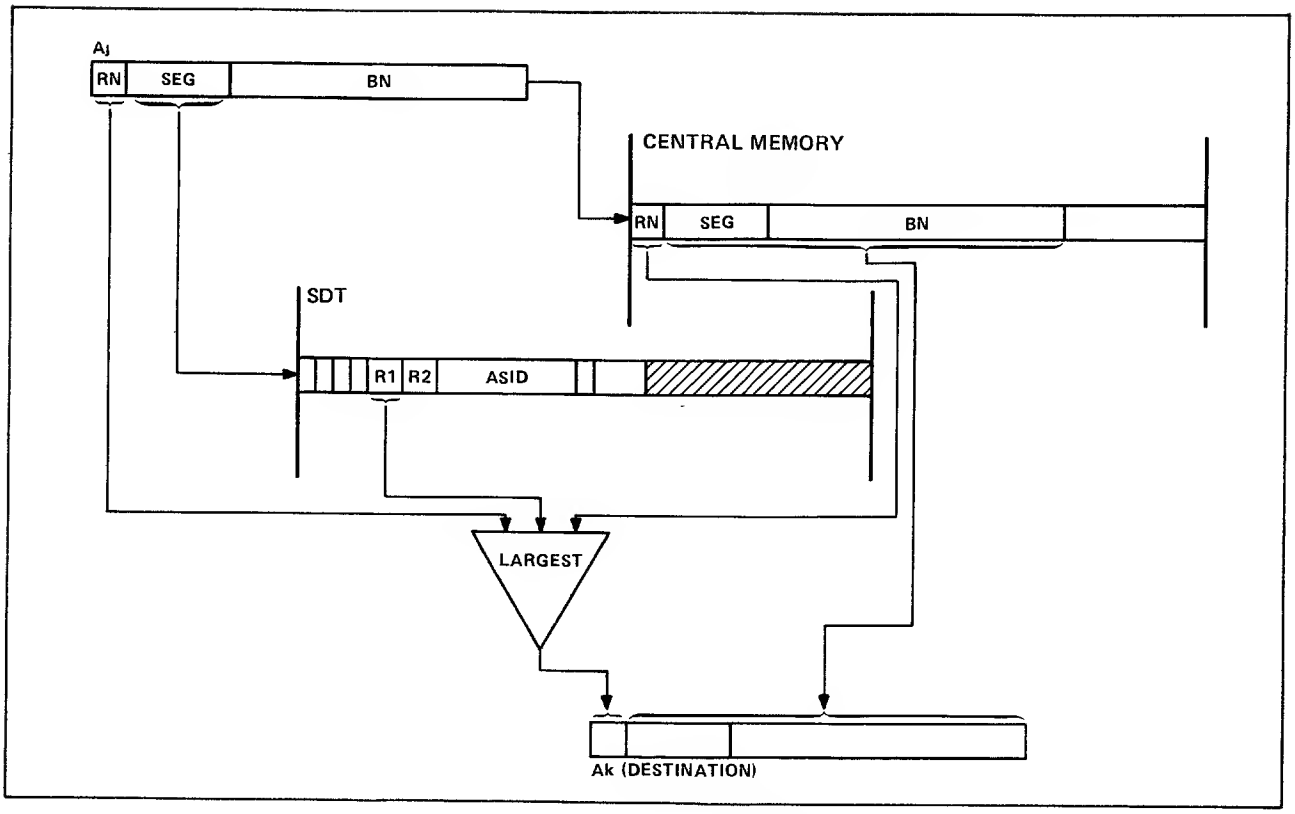


Figure 3-9. A Register Ring Voting

KEYS/LOCKS

Keys and locks provide another hardware protection mechanism. The mechanism restricts access to data that is owned by a procedure or procedures. The following paragraphs contain an example of the key/lock mechanism, a description of the key/lock hardware mechanism, and an example that summarizes the entire security concept.

Key/Lock Use

An example of key/lock use may be taken from a math function that has been developed and is being marketed by some organization. Since the function is general purpose it will typically reside in the same ring of execution as the user who is calling it. However, the developer may wish to restrict access to coefficients he or she has derived and that exist in a separate (read-only) segment. If keys/locks are applied to all segments in the ring, a read-only data segment can only be accessed from a given code segment or segments in the ring.

Key/Lock Hardware Mechanism

There is a lock associated with every segment. It is described by a 6-bit field in the SDE. Up to 64 different locks can coexist. Whenever a segment is executed, the lock associated with that segment becomes the current key. The various values assigned to locks assume no hierarchical significance, as with rings. It is only important if the keys/locks are the same or different. The format of SDE bits 33 through 39 is shown in figure 3-10.

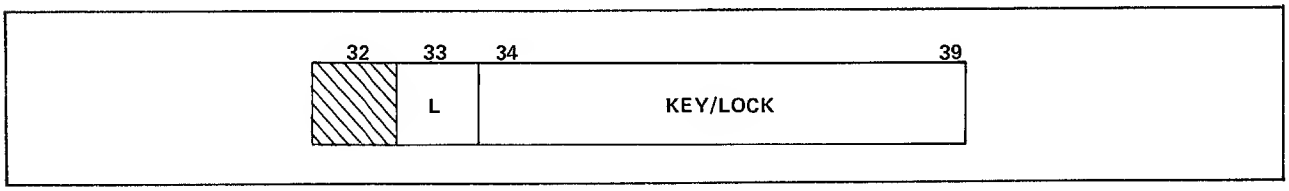


Figure 3-10. Format of SDE Bits 33 through 39

The L field has the following meaning.

Procedure	Local
0	Master key
1	6-bit key

Data	Local
0	No lock
1	6-bit lock

A master key fits any lock, and any key fits a no-lock. In general, access to one segment from another segment is granted only if the first segment has no lock, if the second segment has a master key, or if the key exactly matches the lock. These tests, which are executed by the hardware, are in addition to those already described for rings and type of access. However, the key/lock tests are performed selectively as controlled by the RP and WP fields in the SDE. Even though a lock may have been specified for a segment, the test for read access only applies when the lock applies to read access, as indicated by an RP value of 01. Write accesses are similarly controlled by WP.

On a call, the new key is always taken unconditionally from the callee's key value in the SDE. On return the hardware verifies that the key obtained from the SFSA exactly matches the lock taken from caller's SDE.

Key/Lock Example

Figure 3-11 illustrates the use of key/lock values.

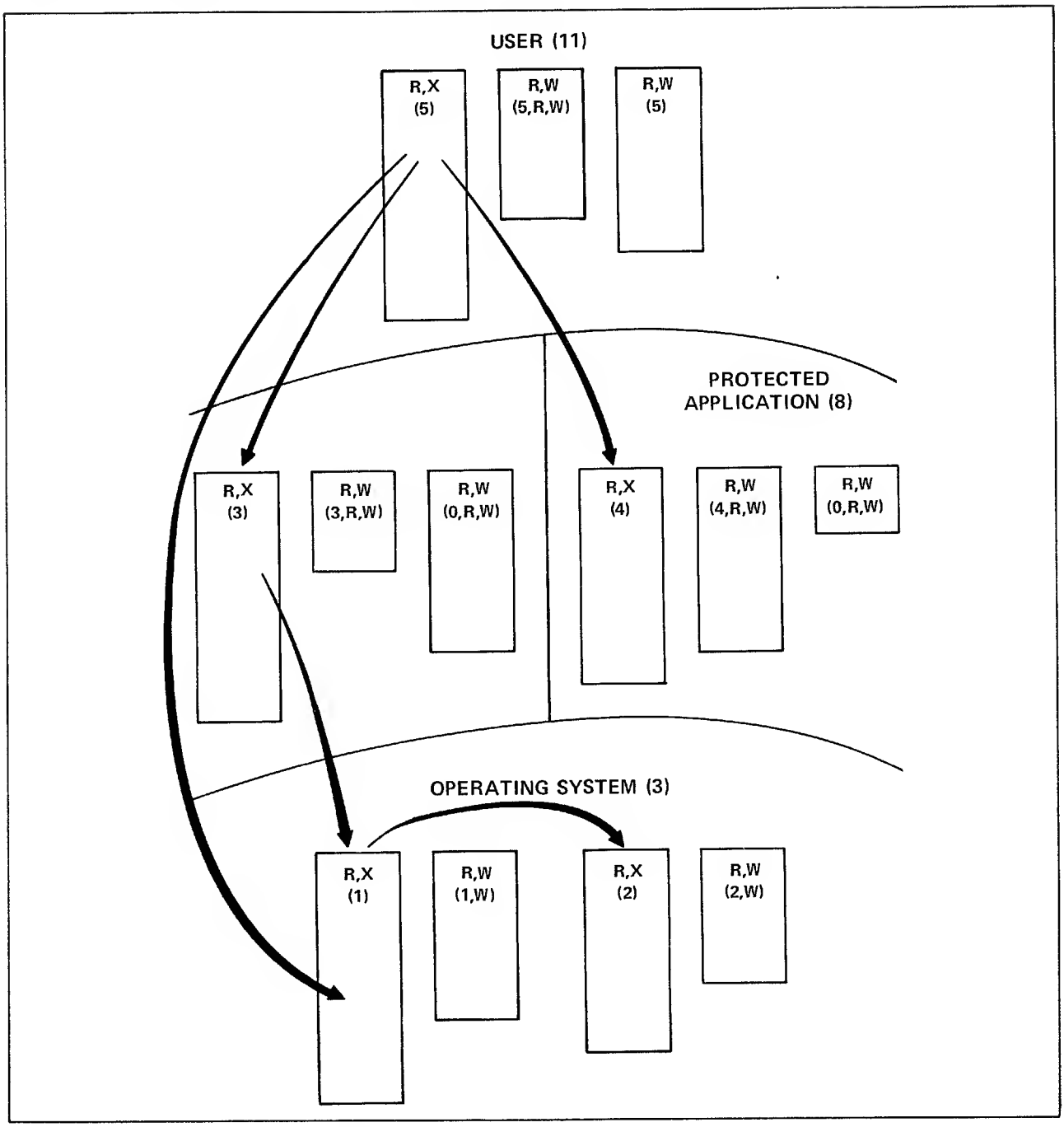


Figure 3-11. Example of Key/Lock Use

In the example in figure 3-11, there are three rings of protection. Ring 3 is the most privileged ring, where parts of the operating system reside. Ring 8 contains two applications that are callable from a user in ring 11. The information in parentheses defines the lock and whether this lock applies to read accesses (R), write accesses (W), or both.

Whenever a call is made to a procedure in another segment, the callee executes with his or her own key. This value is associated with the lock of the data segments accessed by that procedure. In the example, the user has a read/write data segment to which key/lock verification applies. It has a unique key/lock value of 5. Consequently, the applications in ring 8, which have key values of 3 and 4 respectively, and the operating system, which executes in ring 3 with a key value of either 1 or 2, cannot access this data segment. This is true even though the data segment is available for reading and writing and resides in ring 11, a ring with very little privilege. Similarly, the operating system cannot read or write either of the applications' data segments, since they have different key/lock values from the operating system and from each other. Key/locks are used to protect local data regardless of the ring structure in use.

By software convention, the operating system segments (both code and data) are assigned nonzero key/locks. This has the added advantage of protecting various modules of the operating system from each other. In the example, there are two modules, both in ring 3, which can call each other and can read each other's data segments. However, the data segments can only be written from the module to which they belong. This is a very powerful debug aid for the operating system. In other systems, it is not uncommon for one module of the operating system to accidentally destroy data belonging to another module. The damage is not discovered until the second module is called, by which time the culprit is unidentifiable. Through the use of keys/locks, the culprit can be identified both at the time the data is over-written and/or at the time the overwrite is attempted.

The current key is maintained in the P register. Figure 3-12 indicates the 6-bit field for this purpose.

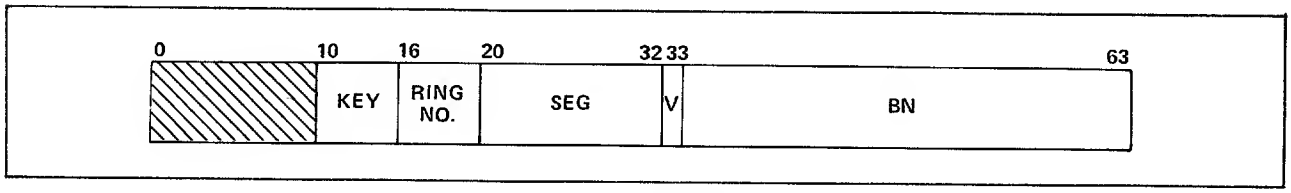


Figure 3-12. Program Address Register

In summary, there are three basic forms of protection from within a user space: the type of access to a segment, the ring protection mechanism, and key/lock values. For every access attempted, all three of these tests must be successful. If any one of them fails, an access violation interrupt results, and the user is exchanged out of his or her address space into the operating system monitor address space, where appropriate action results. The complete, protected user address space is illustrated in figure 3-13.

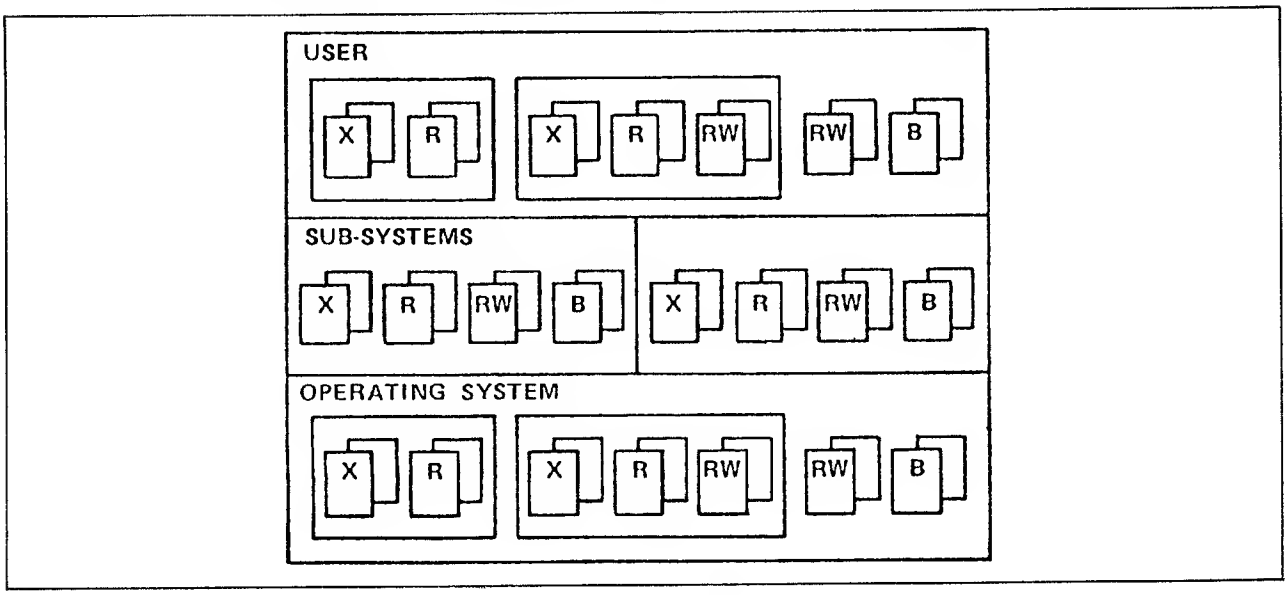


Figure 3-13. Conceptualization of a User Address Space

The following flowcharts (figures 3-14 and 3-15) describe the complete virtual memory address translation and access control.

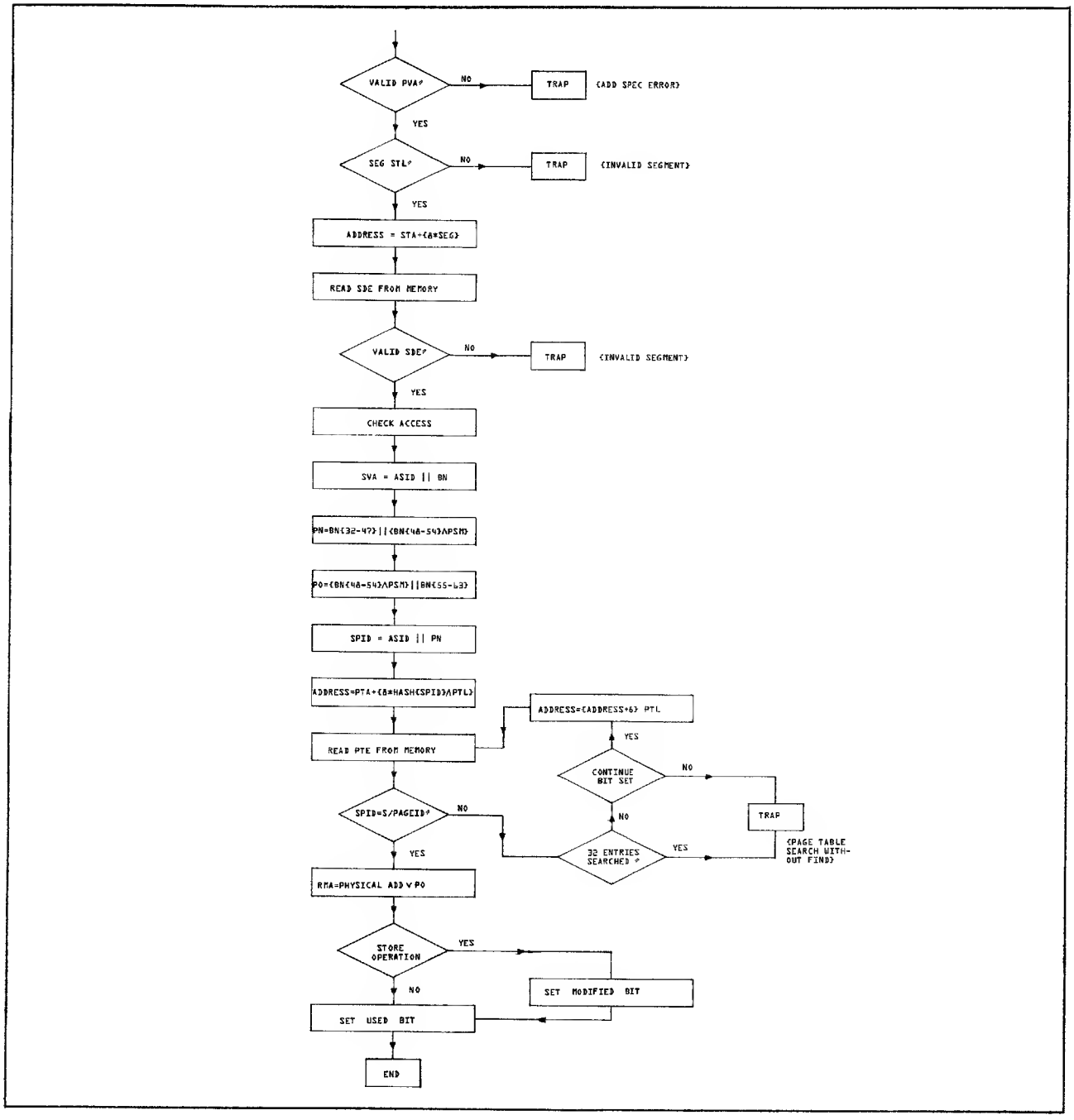


Figure 3-14. Virtual Memory Address Translation Flowchart

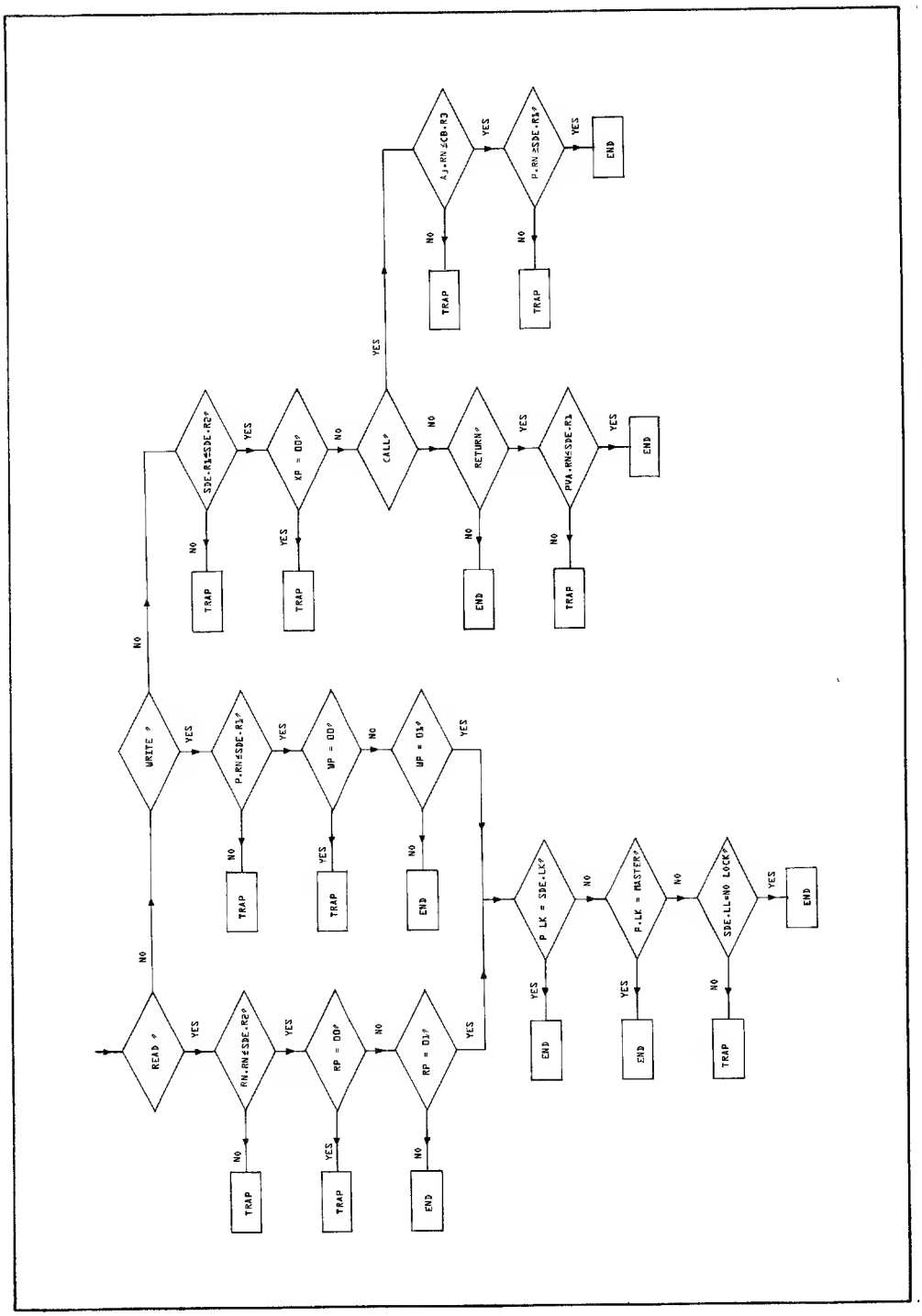


Figure 3-15. Virtual Memory Protection Flowchart

To minimize the time necessary to translate a PVA to an RMA, a number of hardware buffer memories are used. The description given here is based on Model 855 buffer memories. The organization varies from processor to processor, but the fundamental concepts are the same.

Figure 4-1 is a pictorial representation of these buffer memories. The segment map contains the most recently used entries from the process segment table. In the first stage of address translation, the processor uses this map to translate the PVA to an SVA. This SVA is then transmitted to the cache memory and the page map. Each of these buffers is organized on the basis of the SVA, the page map containing the most recently used entries in the SPT, and the cache containing the most recently used words in system virtual memory. Simultaneously, a search is made of the page map and cache. If a cache hit occurs, then no further action is required. However, if the required data is not in cache, then the search of the page map is relevant. If a hit occurs, the required address translation completes and central memory can be accesssed via the appropriate RMA. Only when there is no hit in the page map must the processor actually search the SPT in real memory.

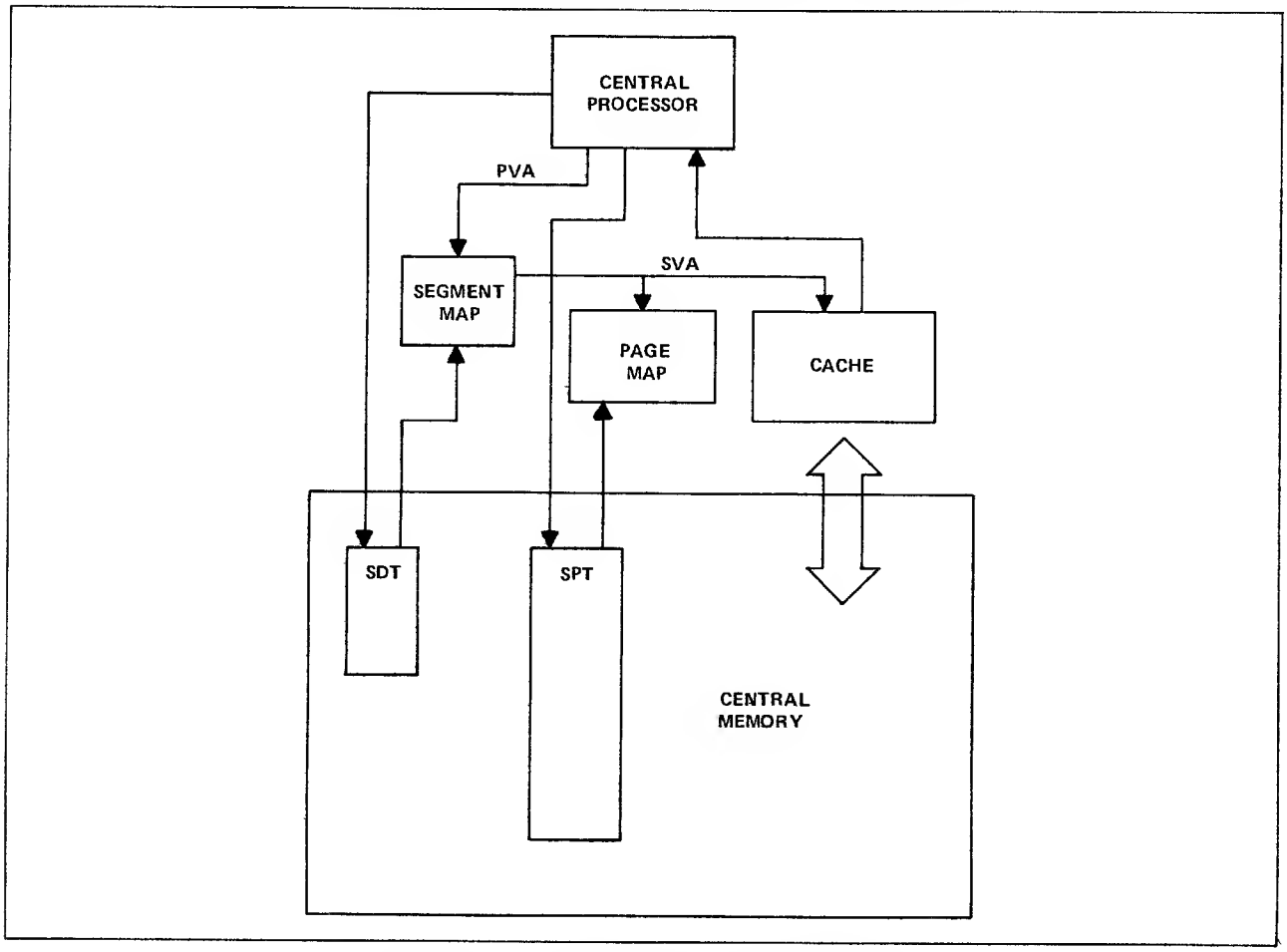


Figure 4-1. Virtual State Buffer Memories

SEGMENT MAP

The purpose of the segment map is to translate a segment number (SEG) to an active segment identifier (ASID). This is the first step in the address translation mechanism and translates a PVA to an SVA. Figure 4-2 illustrates the general process. A set-associative technique is employed, where an index is used to select a set, and an associative (simultaneous) comparison is made between each entry in the set and the required segment number. Model 855 has 16 such sets in its segment map, each set having two members. To index into the map, the lower 4 bits of the segment number are used as a hash index. These bits are the most random part of the segment number.

The hash index identifies one of two entries in the segment map that are candidates for translation of the given SEG. The segment map simultaneously compares the set-tag entries with the mode of operation (job/monitor) and the upper 8 bits of the SEG. If a hit is made, the ASID is taken from the segment descriptor word held in the map. If no hit occurs, the ASID must be fetched from the segment descriptor table (SDT) in real memory.

The tag field of the segment map contains a bit to indicate the entry in the two sets that is the least recently used (LRU). There are only two candidates. This entry is used to receive the new segment descriptor. The tag field does not contain the segment table address (STA). Instead two registers are used, one for the job STA and one for the monitor. During an exchange to monitor state, the monitor STA is compared with the STA obtained from the monitor exchange package. Similarly, the monitor STA is compared with the job STA when an exchange to job state occurs. If the values do not compare, all entries for either job or monitor in the segment map are invalidated.

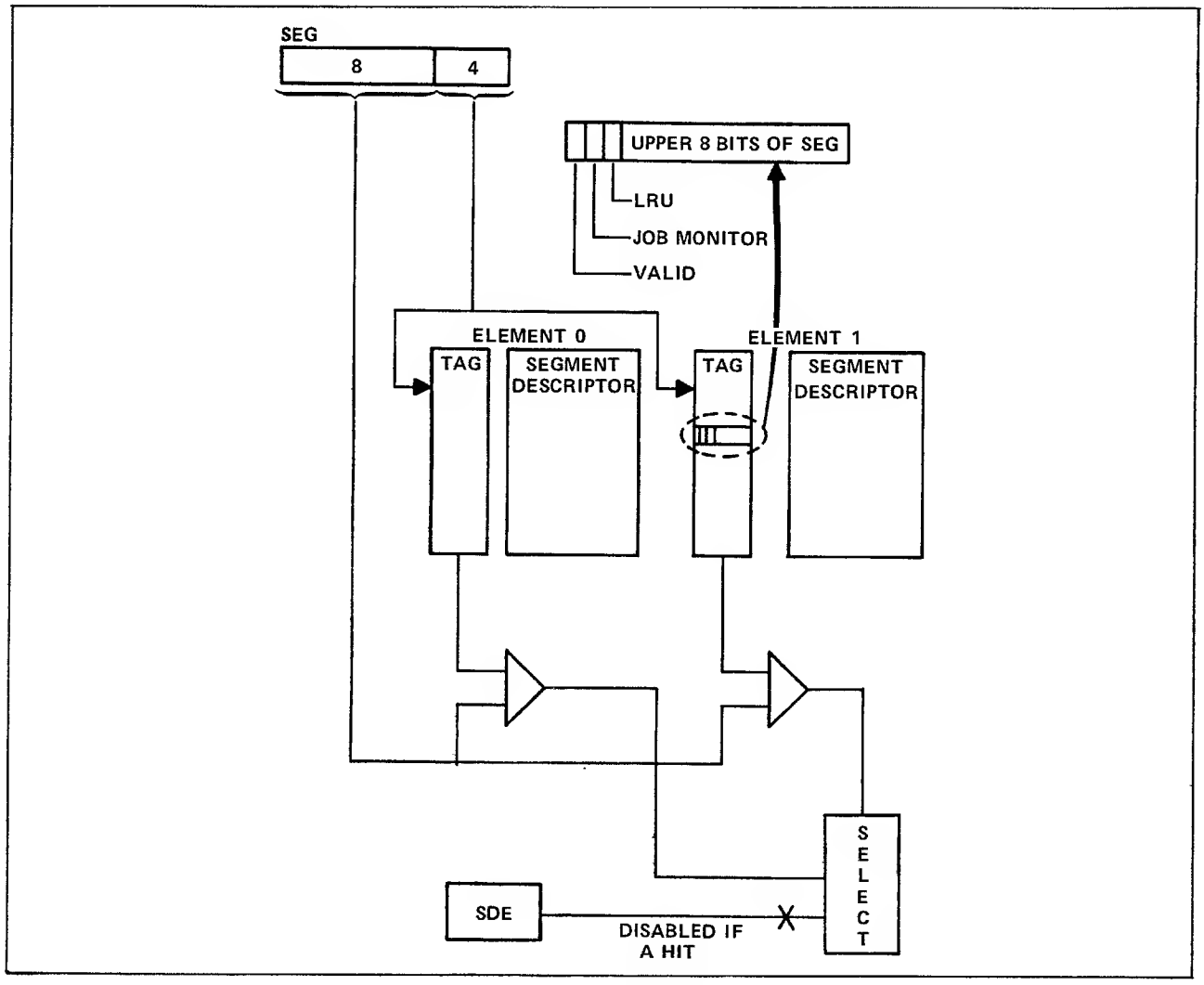


Figure 4-2. Segment Map Operation

The most recently used segment numbers appear in the map. The more segments used by a process, the less likely it is to find the entry in the map. The system performs most efficiently if the map entries for monitor and job are not hashed to the same location in the map. This is best handled by the operating system assigning job segment numbers sequentially from zero and monitor segment numbers sequentially from FFF downwards. This has the effect of creating a 32-entry buffer filled from the top with job segment descriptors, and from the bottom with monitor segment descriptors (figure 4-3). The choice of a starting segment number for monitor need not be FFF, but should be of the form XXF. In fact, FFF is probably used for some special purpose by the operating system, and, in any case, maximizes the dead space in the monitor segment table. The practical choice for the starting segment number is computed from:

(number of monitor segments) .OR. 00F

in which case the maximum number of dead entries is 15.

When the map is degraded (due to a parity error), one set is eliminated. The probability of a miss is heightened and performance degrades.

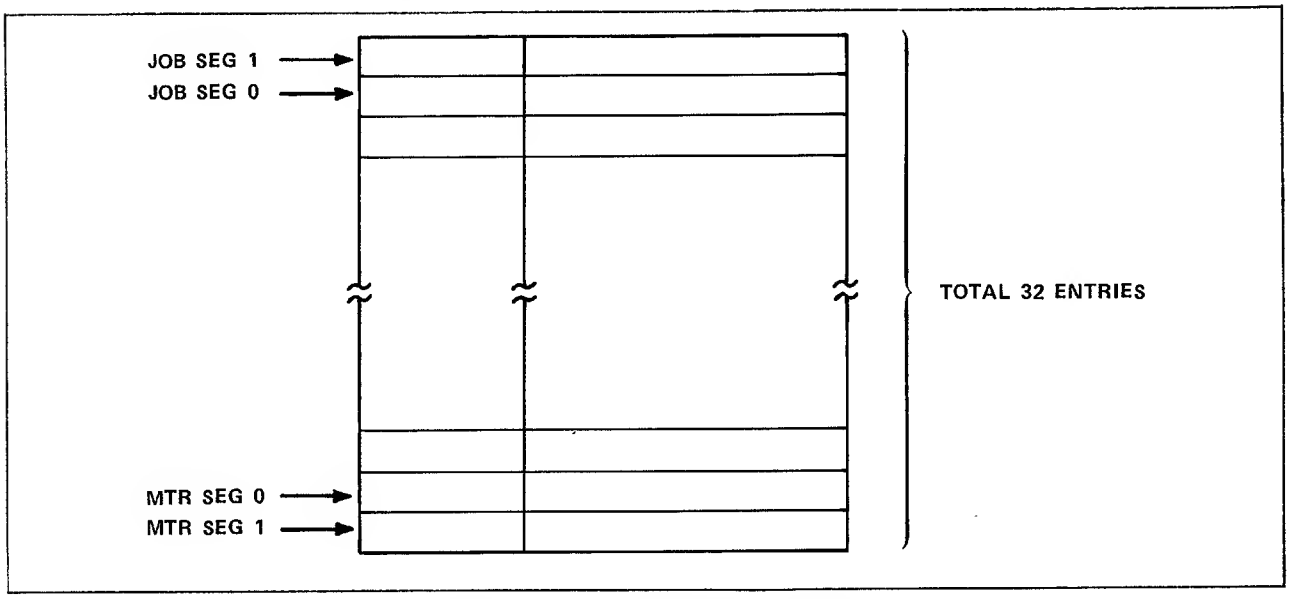


Figure 4-3. Segment Map Allocation

PAGE MAP

The purpose of the page map (figure 4-4) is to translate the SVA from the segment map into an RMA. As with the segment map, a set-associative technique is used. In this case there are 32 sets, and the low-order 5 bits of the page number are used as a hash index to select a set. The page number is formed from the byte number by executing a logical product with the page size mask (PSM). Depending on the page size, the page number is not right-justified, and the hardware performs the necessary justification before extracting the hash index.

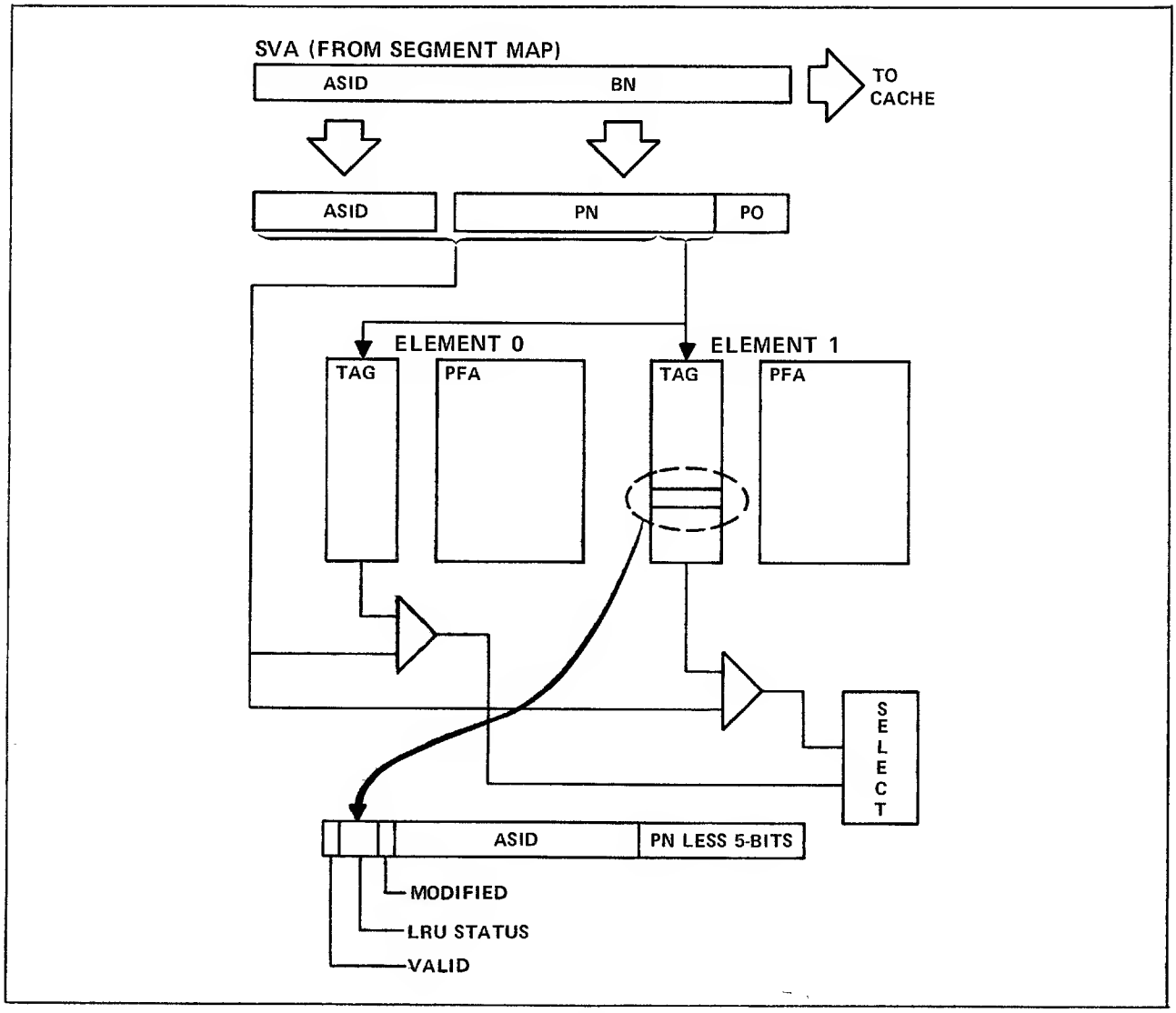


Figure 4-4. Page Map Operation

The page map simultaneously compares the set-tag entries with the high-order 33 bits of the SVA. The valid bit is not included in this operation. Invalid PVAs (and therefore invalid SVAs) do not get this far in the translation mechanism. If a hit is made, the RMA is formed from the SVA in the page map data table, and the page is offset. Otherwise, a page table search is initiated.

The tag field in the page map contains 2 bits to indicate which entry in the two sets is the LRU. There are only two candidates. However, 2 bits are allocated on Model 855 to allow for up to four entries per set.

The most recently used pages appear in the page map. The more pages used by a process, the less likely it will be to find the entry in the map. When the page map is degraded, one group of entries is eliminated. The probability of a hit is reduced, and performance degrades.

The modified bit is carried in the page map, but not the used bit. Descriptions of actions taken on a hit and a miss follow, and clarify the setting of these bits.

- Map Miss - Page Table Hit

  Read (i) Set the used bit in the PTE.
  (ii) Copy the modify bit to the map.
  (iii) Copy the addresses to the map.

  Write (i) Set used and modify bits in the PTE.
  (ii) Copy the modify bit to the map.
  (iii) Copy the addresses to the map.

- Map Hit

  Read (i) Simply form the RMA - no page table access is necessary.

  Write (i) If the modify bit is set in the map, the process is identical to read.

  (ii) If the modify bit is not set in the map, a page table search is required to set the modify bit in the PTE. The same bit is set in the map. At this time the modify bit in the PTE and in the map is set, and the used bit is set in the PTE.

The map and the cache perform similar functions. Once the segment map has formed an SVA, the segment map sends the SVA simultaneously to the page map and the cache. If there is a cache hit, and the operation is a read, there is no need to access the page map, as data is being read directly from cache.

On a read, the cache hit overrides everything. It is possible to to get a cache hit even when the relevant page is not in central memory, since the cache is organized on the SVA. The operating system must ensure that cache accurately reflects the contents of system virtual memory at all times. Whether the data actually resides on disk or in real memory is immaterial. On writes, the situation is different. Virtual State processors always write through cache. The appropriate entry in cache is either updated or purged on a write. Actual implementation is processor-model dependent. On CYBER 170 Model 855, when a cache hit occurs on a write, if it is a full-word write the word is updated. For a partial-word write, the word is purged. On another processor, cache is updated regardless of the nature of the write.

CACHE MEMORY

Virtual State supports very large, cost-effective memories. It achieves this at the expense of some memory speed. To make up for this loss of speed, a buffer memory (cache memory) is placed in the faster processors. The most recently used words in system virtual memory are held in a much smaller, faster memory. The management of this memory is shown in figure 4-5. A set-associative technique is used to control entries in the cache. On Model 855, a maximum of four entries per associative set is employed. An entry in a set consists of a tag field that identifies the entry and 32 bytes (4 words) of data called a block. There are 256 sets on Model 855.

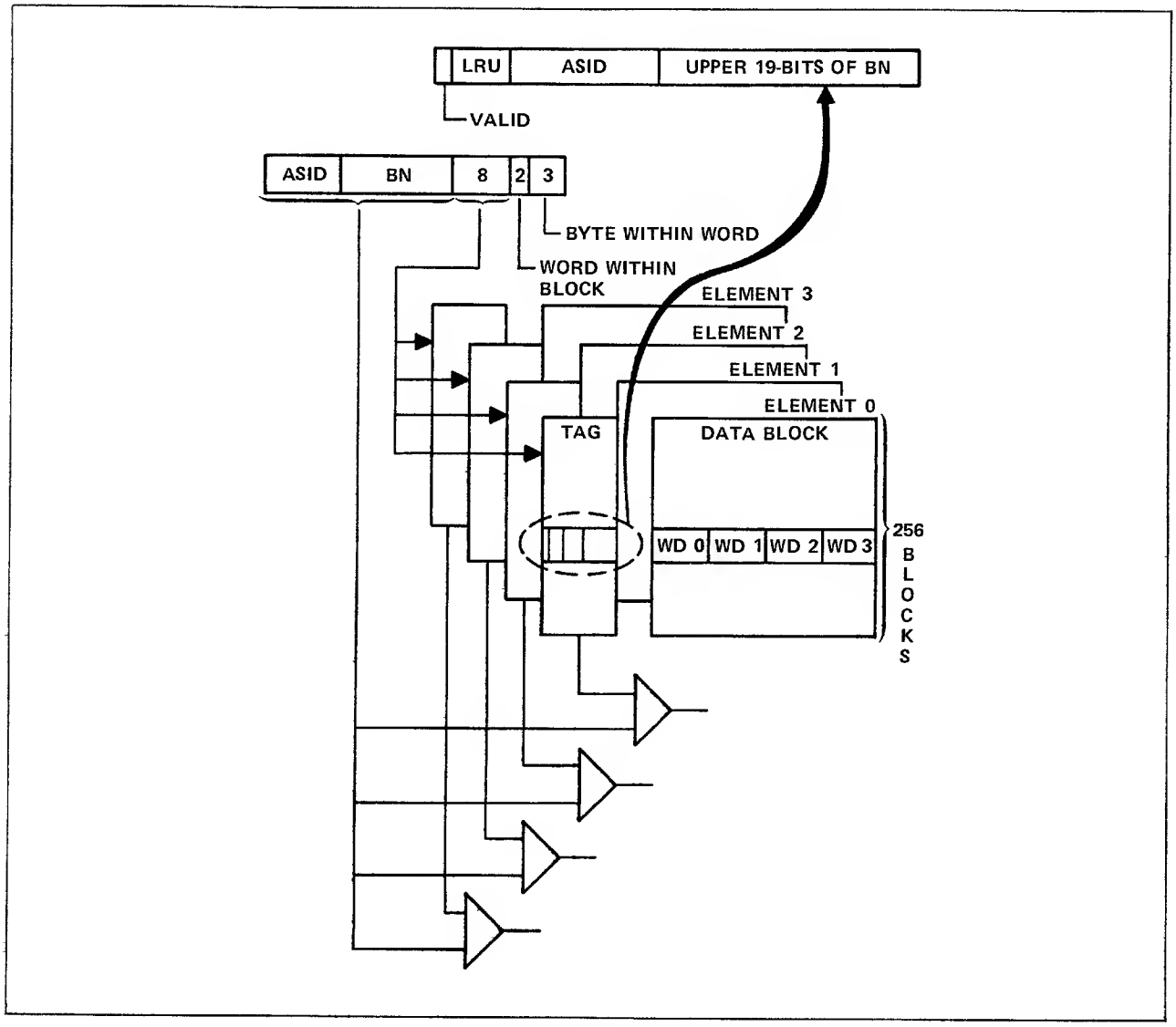


Figure 4-5. Cache Memory Operation

Bits 51 through 58 of the SVA are used as a hash index into the sets. These bits represent the most random part of the SVA. The low-order 5 bits of the SVA represent the word within block and the byte within word, respectively. The ASID does not enter into the hash index computation. This is deliberate, since in CYBER 170 State only a single segment (ASID = FFFF) is used, and this segment has no randomness.

Once a set has been selected, a simultaneous comparison of the upper 35 bits of the SVA and the tag entries is made. If there is a hit, and the entry is valid, that entry is used. If there is no hit, a set is chosen for the new entry and the appropriate words read up. Entries are chosen first on the basis of their validity and second on their LRU status. Whenever a new entry is made in a set, an entire block (four words) is read up, starting with the required word and proceeding from left to right, unless the instruction is a right-to-left (BDP numeric) type. Cache regards central memory as a series of four-word blocks that always start on a block boundary.

If, at any time, cache is not busy after it finds a hit, it automatically looks ahead one block. If it gets a hit, then the sequence ends. If cache doesn't get a hit, it initiates a read on that block.

Cache is always organized on SVA for Virtual State processors.

SOFTWARE IMPLICATIONS

There are several software implications in the use of the cache and the maps, particularly in a multiprocessing environment. The operating system software must ensure that stale data does not exist at any time in the map or the cache (Virtual State physical I/O and memory writes performed by another processor do not automatically update the processor-local cache). The following guidelines should be followed by the software.

- Whenever a page table entry is changed the page maps must be purged, both the page map in the processor updating the page tables, and the page map in the second processor, if available. Care must be exercised by the software at this time. The hardware depends on the software to take certain precautions, since the Virtual State instructions are not interruptible. Before an instruction is placed in execution it is prevalidated. The hardware ensures that all pages required to complete the execution of the instruction are in memory before execution begins. Once execution starts, the processor assumes that the pages it requires are there. A second processor must not delete a page from memory without first notifying the other processor. A typical sequence of events is:

    1. Set the invalid bit in the PTE. This ensures that an instruction requiring this page cannot start; it can complete if it has already started. In other words, the processor ignores the valid bit once an instruction has been prevalidated.

    2. Send an interrupt to the second processor asking to purge map.

    3. First processor waits for acknowledgement from second processor that map has been purged.

    4. First processor updates the PTE.

    5. First processor sets valid bit in PTE.

Since the valid bit was dropped prior to sending the interrupt, no instruction can be started using the absent or deleted page. An instruction making such a reference causes a page fault, and this page fault is not processed until the in-progress page table update is completed. This is another interlock that must be set up by the operating system software. Only one processor can execute a page table update at one time.

When a page table update is made, cache memory need not be purged if the operation is a write. Since writes always write through cache memory, prevalidation ensures that the page exists in memory. If the operation is a read, even though the page has been purged from memory, the copy in cache memory is still good and the hardware uses this copy, as has already been described.

- There is a danger, in a multiprocessor environment, of the cache becoming stale whenever a processor is assigned to a job. At this time, the operating system should check the last processor identification (LPID) field in the job exchange package against the processor identifier (PID). If the quantities are not identical, the cache must be purged.

These are not the only times when cache and the map must be purged. Similar problems arise during input/output and are discussed in volume II of the Virtual State hardware reference manual (refer to the preface for more information).

This section discusses processor state and process state registers. Processor state registers define the operational state of the processor without regard to a specific process. Process state registers define a specific process.

PROCESSOR STATE REGISTERS

Each processor has a set of registers that define the operational state of the processor. These registers are described fully in the model-independent general design specification (MIGDS), however, several points are of interest here.

- Virtual State has an exchange mechanism, similar in function to CYBER 170 State, that executes quite differently from CYBER 170 State. Whereas on CYBER 170 State a true exchange occurs (the operating registers are stored in memory and loaded with the contents of those same memory cells), on Virtual State the operating registers (process state registers) are stored in one area of memory and loaded from a different area in memory. Since an exchange jump always changes the operating mode from job to monitor, or vice versa, two exchange packages are located in memory, a monitor exchange package and a job exchange package. These exchange packages are located at RMAs specified by the job process state (JPS) and the monitor process state (MPS) registers. The exchange packages must not be located at the same address, nor must they overlap. Finally, the packages must be on a double-word boundary. For this reason, the least significant 4 bits of the JPS and MPS are ignored (treated as zeros, as shown in figures 5-1 and 5-2).

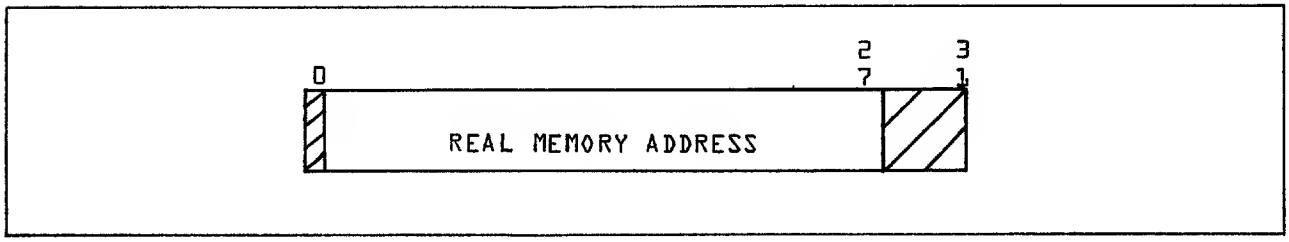


Figure 5-1. JPS and MPS Registers

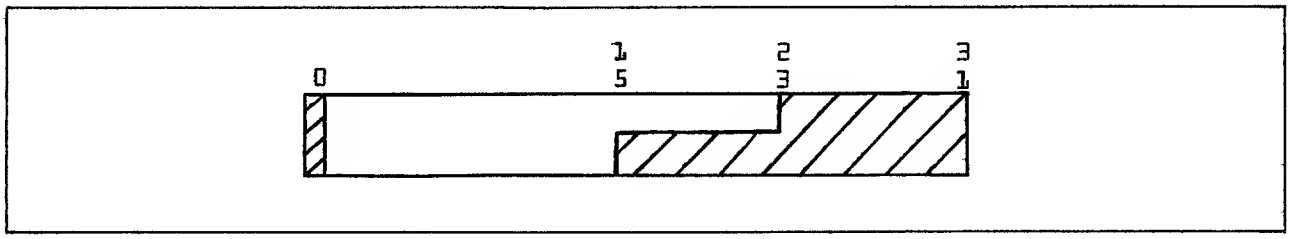


Figure 5-2. PTA Register

- Two registers, the page table address (PTA) and the page table length (PTL), specify the size of the page table. The page table must be located on a boundary that is zero modulo the page table length, since the hardware accesses the page table frequently and computes an index for this purpose. To find the address of the required entry, the index is catenated to the PTA, a much faster operation than adding the index to the PTA. Depending on the page table length, the low-order 9 through 17 bits of the PTA must be set to 0.

  The PTL, which indicates the length of the page table, is used as a mask to ensure that a hash index with the page table remains within the bounds of the page table. Its use is described in the section dealing with virtual memory.

- The page size mask (PSM) specifies the page size to be used. The page size may be from 512 bytes to 64K bytes. However, typical page sizes are expected to be 2KB and 4KB. As with the PTL, the use of the PSM is discussed fully in the virtual memory section.

- Two registers deal with equipment identification. These registers are the element identifier (EID) and the processor identifier (PID). The first is a unique, world-wide identification. The format of the EID is shown in figure 5-3. The second is an abbreviated version that uniquely identifies an equipment within a system. The PID is used on exchanges to identify the last processor identification (LPID), and is used in a self-discovery process during system initialization. A third register, options installed (OI), completes the description of the equipment. This is a 64-bit register that indicates the number of PPs, cache memory size, ports to central memory, and so forth.

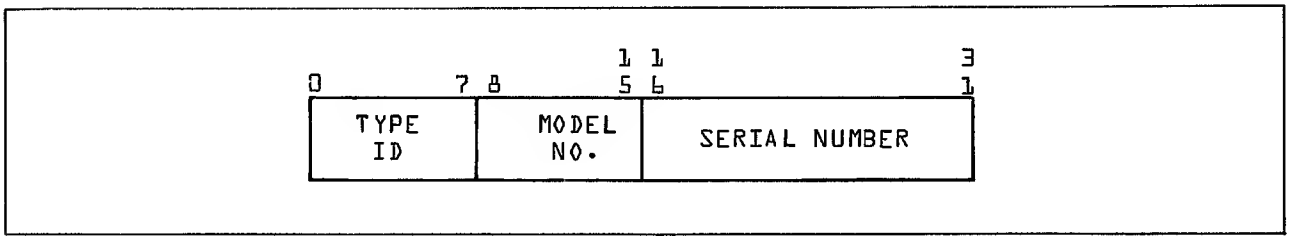


Figure 5-3. EID Register

- A 32-bit microsecond counter, the system interval timer (SIT), counts down and is used to establish job time slices.

- One final register of interest is the virtual machine capability list (VMCL). Many of the Virtual State processors are microprocessors and the microcode may describe various machines termed virtual machines. Virtual State is one such virtual machine, but many others are possible, in particular CYBER 170 State. This 16-bit register controls the virtual machines the user (customer) is permitted to run. For example, a Virtual State customer who has not purchased the CYBER 170 State emulator is prevented from executing CYBER 170 State code via the register.

The remaining processor state registers (there are several) deal with the operational status of the processor and its maintenance. Many of these registers are model dependent.

Access to these registers is controlled. Most registers can be read and written from the System Monitor Utility (SMU), and can be read from the processor. However, registers can only be written when the appropriate privilege has been granted. Access to the registers is illustrated in figure 5-4.

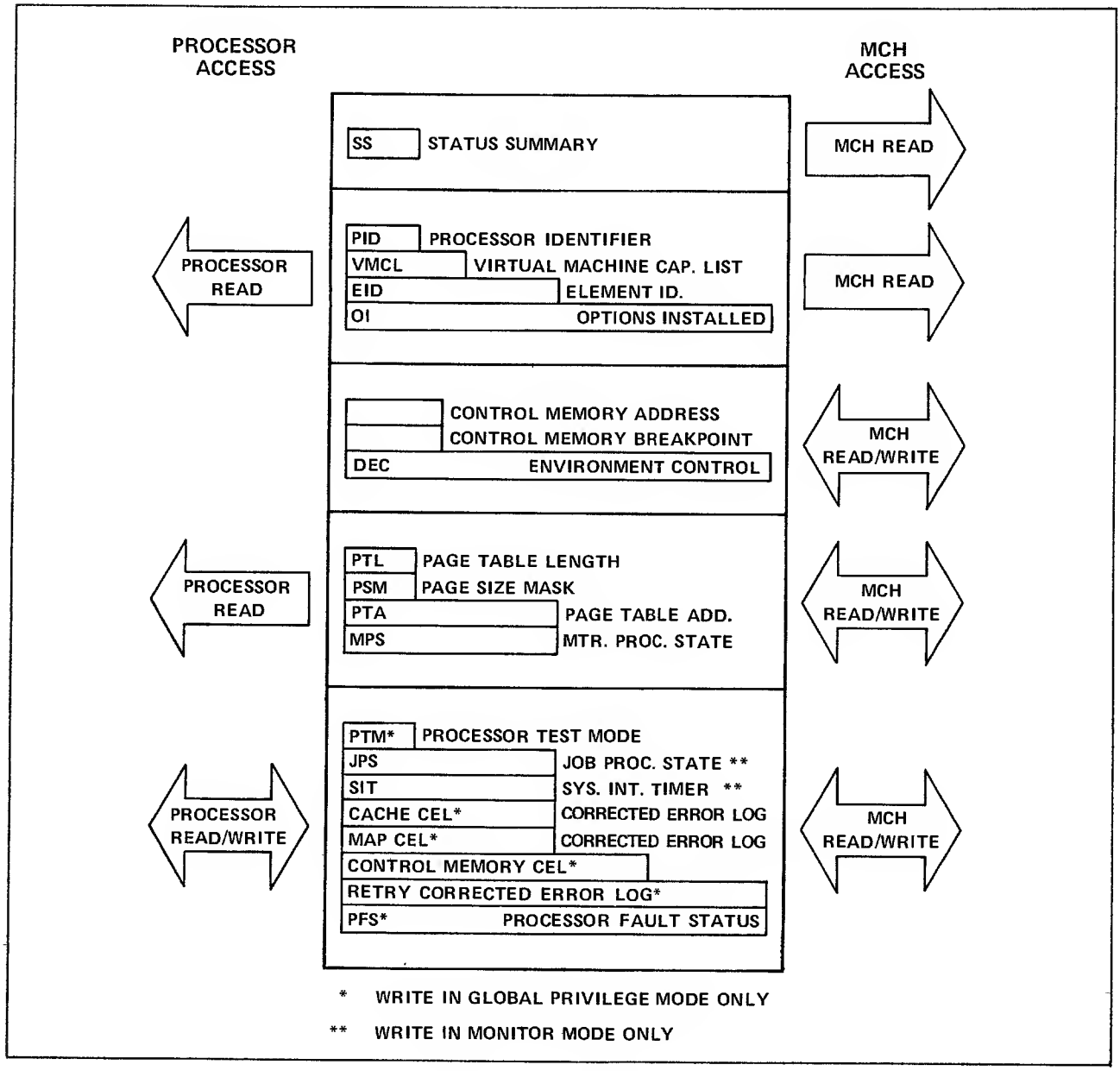


Figure 5-4. Processor State Registers

PROCESS STATE REGISTERS

There is a large set of registers that define each process state. Included are the P, A, and X registers. These registers completely describe the operational environment of a job or process. If the process is interrupted for any reason, that operational environment must be captured in order for processing to resume after the interrupt has been dealt with. This is accomplished by the exchange mechanism, which saves all the process state registers in an exchange package (figure 5-5) and loads from a second exchange package a fresh set of registers that define the process exchanged to.

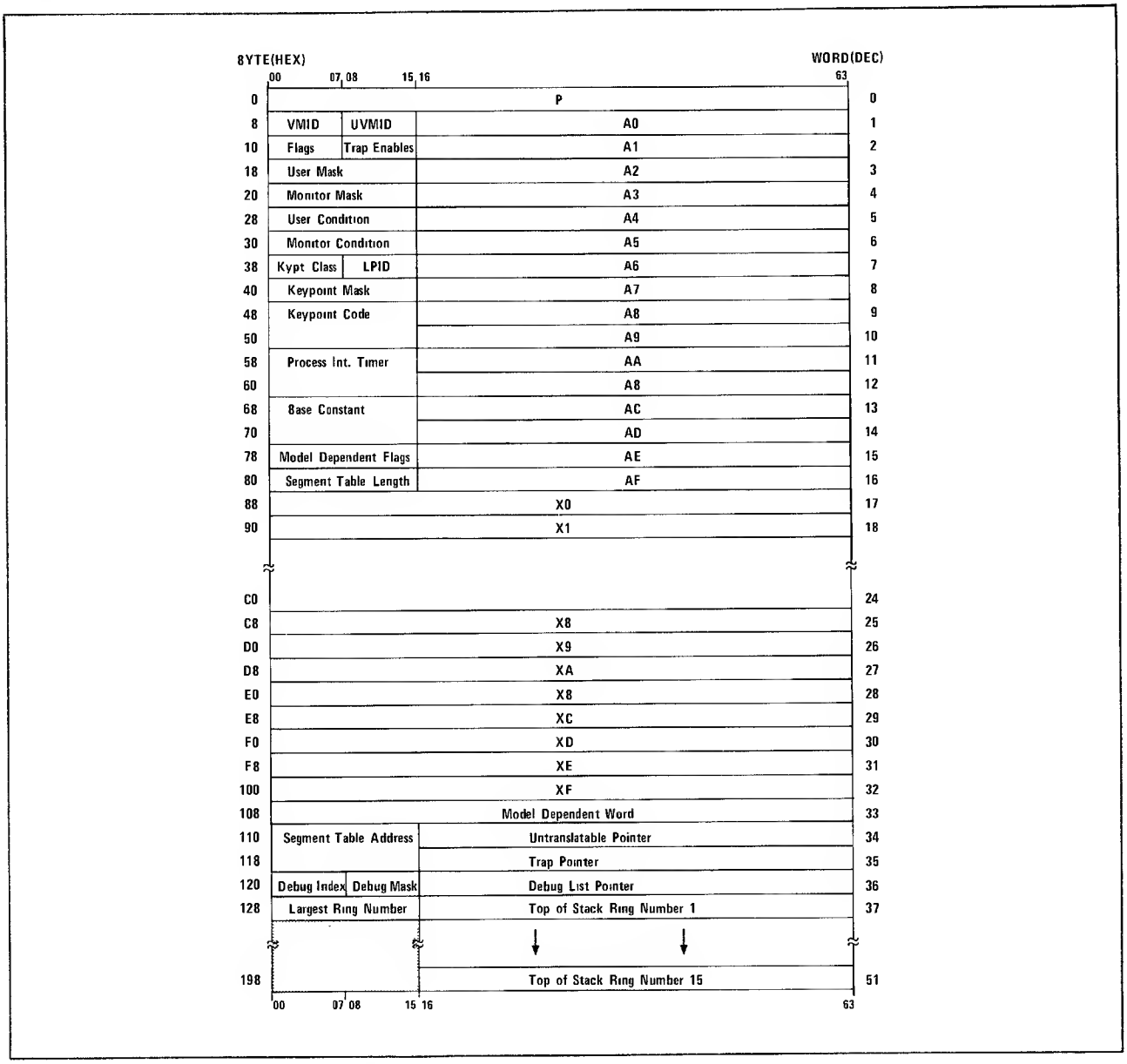


Figure 5-5. Virtual State Exchange Package (Virtual State Process)

The 33 basic operating registers (P, A, and X registers) are described in section 1 of this manual. The following summarizes the remaining process state registers.

- The virtual machine identifier (VMID) designates the virtual machine to which control is being transferred. VMIDs of 0 (Virtual State) and 1 (CYBER 170 State) have been defined for the Virtual State processors. The UVMID is a register used to designate an invalid (undefined) VMID to which the processor attempted to transfer control. If an exchange jump is attempted to a nonexistent virtual machine, the exchange completes, and a second exchange interrupt occurs immediately on an environment specification error. The UVMID is then set to identify the fault to the operating system.

- A series of flags is located in word 2 of the exchange package. The flags are the critical frame flag (CFF), the on-condition flag (OCF), the keypoint enable flag (KEF), the processor not damaged flag (PND), and two flags to control trap interrupts. These are primarily software flags carried by the hardware. Their use is described later in this manual.

- The user and monitor mask registers and condition registers are used to control interrupts and are discussed fully in the sections dealing with interrupts. Similarly, the keypoint class, keypoint mask, and keypoint code registers are described in the section dealing with keypoint. These registers control the keypoint process.

- The LPID has already been introduced (refer to Cache Memory). It records the PID of the processor executing a given exchange interval. The processor interval timer (PIT) is a 32-bit microsecond timer analagous to the system interval times (SIT). It counts down, at a microsecond rate, and interrupts the processor whenever it reaches zero. It is used for timing within a given task.

- The base constant is a register used by the operating system as an index to a control point area for an executing task. The segment table address and length (STA and STL) specify the RMA and length of the SDT to the hardware. Remember, the SDT is a hardware table used in the virtual memory address translation and, as such, it must be located at a real memory address. The combination of the STA and STL also uniquely defines the task address space.

- The model-dependent flags and word are used by the hardware, typically, to help in hardware checkout. They do not have any particular significance to the software. The debug index, debug mask, and debug list pointer are used to control the debug facility, and are discussed fully later.

- The trap pointer carries the address of the trap handler to be used by an executing task. It is discussed in the sections dealing with interrupts along with the untranslatable pointer (UTP). This register holds the pointer or address that could not be translated, causing an exchange to operating system monitor. There are 15 top-of-stack pointers, one for each ring of execution. Their use is covered in the section dealing with call/return. On most processors, these pointers are not kept in live registers, but instead reside in the exchange package in central memory. The largest ring number register has been included in the event that the top-of-stack pointers are kept in live registers, in this case, the hardware could be organized so that the exchange mechanism only has to exchange those pointers actually in use in the process.

As with the processor state registers, access to the process state registers is carefully controlled. This access is illustrated in figures 5-6 and 5-7.

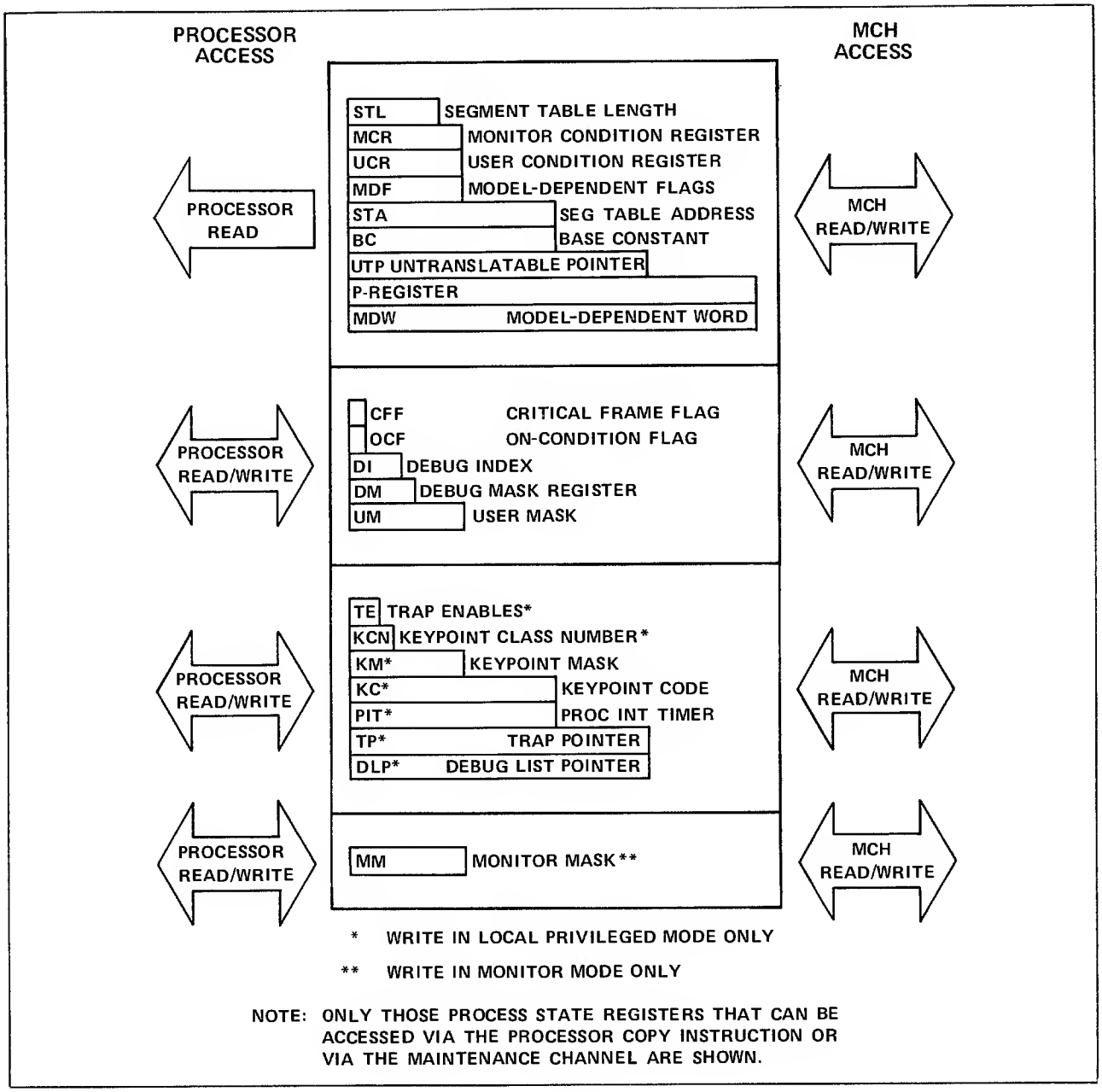


Figure 5-6. Process State Registers

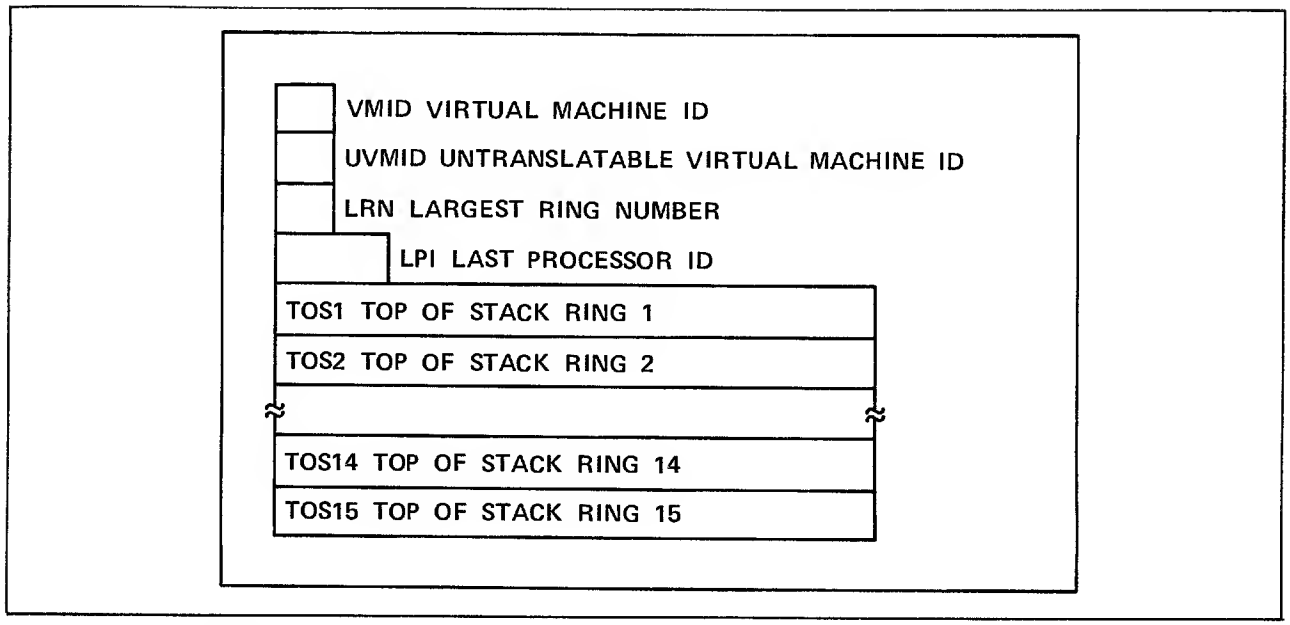


Figure 5-7. Process State Registers Accessed by Exchange Operation

The Virtual State interrupt system is hierarchical. A process can be interrupted and control transferred to an operating system interrupt handler. Depending on the status of this new environment, it may be interrupted via a different mechanism. The two basic interrupt mechanisms are exchange interrupts and trap interrupts. Both forms of interrupt save the current environment (as described by the process state registers) and transfer control to some other code module. In the case of an exchange interrupt, control transfers from a process address space to the monitor address space. Trap interrupts, on the other hand, are processed within the address space of the current process.

Trap interrupts are controlled by two process state registers, the trap enable flip-flop (TEF) and the trap enable delay flip-flop (TED). The settings of these registers are controlled by the exchange mechanism. It is the software designer's choice whether a monitor exchange interrupt is handled with traps enabled or disabled. The importance of this design decision is described later.

Two pairs of process state registers are used to monitor interrupts and to control the actions taken when a condition arises that may interrupt a process. These are the monitor condition register (MCR) and monitor mask register (MM) and the user condition register (UCR) and user mask register (UM). The condition registers are normally filled with zeros. Each bit in the registers corresponds to a particular interrupt condition. When that condition is encountered, the bit is set to indicate that fact. For each bit in the condition registers, there is a corresponding bit in the mask registers. When both bits are set, an interrupt is taken. In other words, the processor takes the logical product of the two register pairs and takes an interrupt if the result is nonzero (figure 6-1).

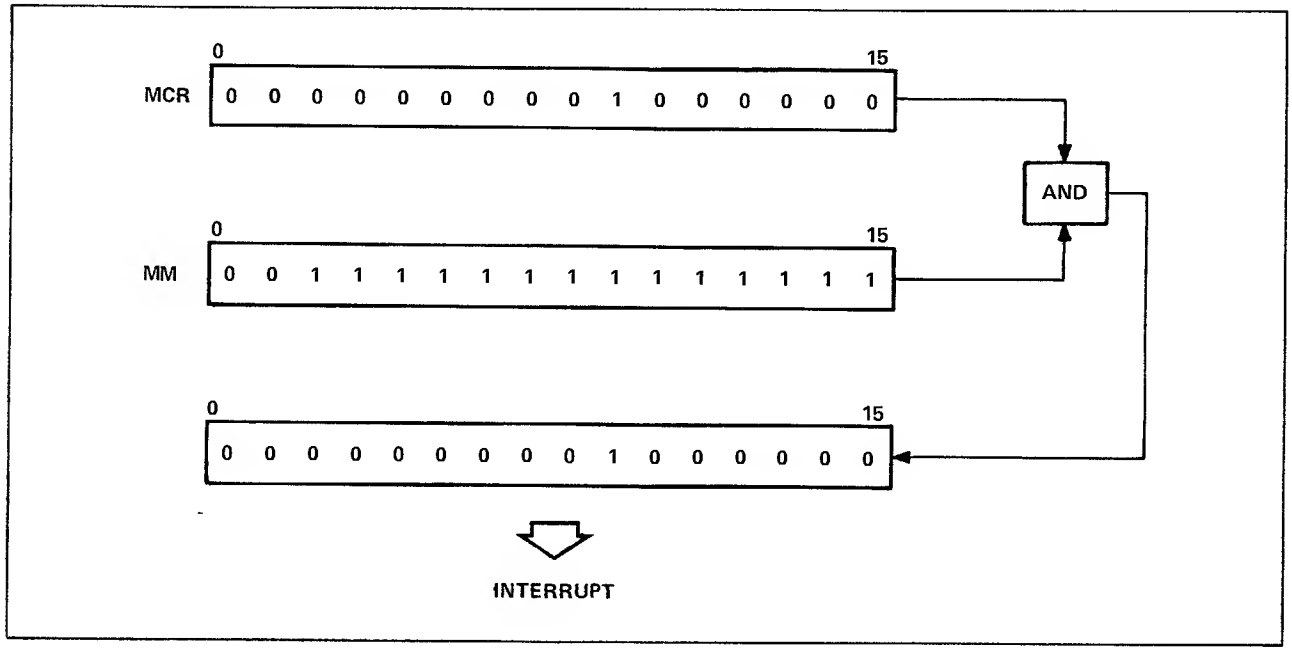


Figure 6-1. Basic Interrupt Mechanism

Although there are only two condition registers (for the monitor and user), there are really four classes of conditions. They have been grouped into two registers for software convenience. The four classes are monitor conditions, system conditions, user conditions, and status indicators (figure 6-2). Conditions signaled in the MCR have a higher priority than (are acted on before) those flagged in the UCR. The MCR contains all system conditions, flags, and most of the monitor conditions. The UCR contains all user conditions and some monitor conditions. The monitor conditions in the UCR are there for the user to process via a trap interrupt from within the user address space.

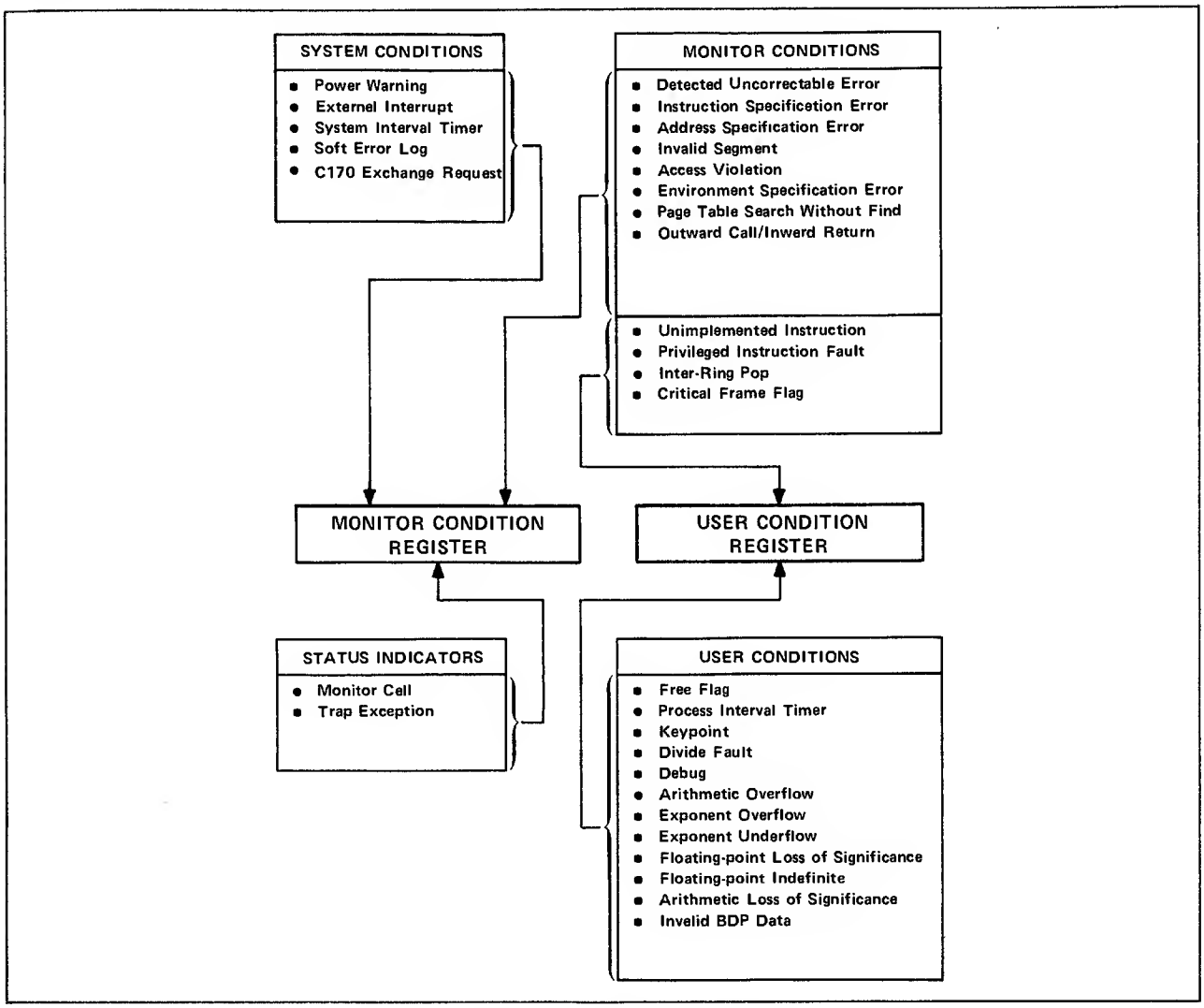


Figure 6-2. Interrupt Conditions

Monitor conditions are organized so they are typically encountered only in job mode, the exceptions being uncorrectable errors that can occur in either job or monitor mode. When these conditions arise, an exchange jump from job mode to monitor mode takes place. A recurrence of the same condition, or another monitor condition, causes the processor to halt when traps are disabled. With the exception of hardware diagnostics, the code executed in

monitor state is arranged so these conditions cannot arise. System conditions, on the other hand, occur any time and cause an exchange interrupt from job state to monitor state. They are stacked when encountered in monitor mode with traps disabled. Care must be taken when processing an interrupt to ensure that conditions are not lost.

Consider the following situation: the machine is in job mode with traps enabled and a page fault occurs (figure 6-3). During the processing of the page fault in monitor mode, a soft error occurs. If traps are disabled, this condition is simply remembered (stacked). When the page fault processing completes, if an exchange is either taken back to the process originally interrupted or to another process, the soft error is lost. It is stored away in the monitor exchange package. There is only one way to guarantee that this condition is not lost and that is to run in monitor mode with traps enabled. Testing the live MCR does not suffice, since subsequent to this test, an exchange back to job state must be made, and there is a finite time between the test and the point where the exchange is committed.

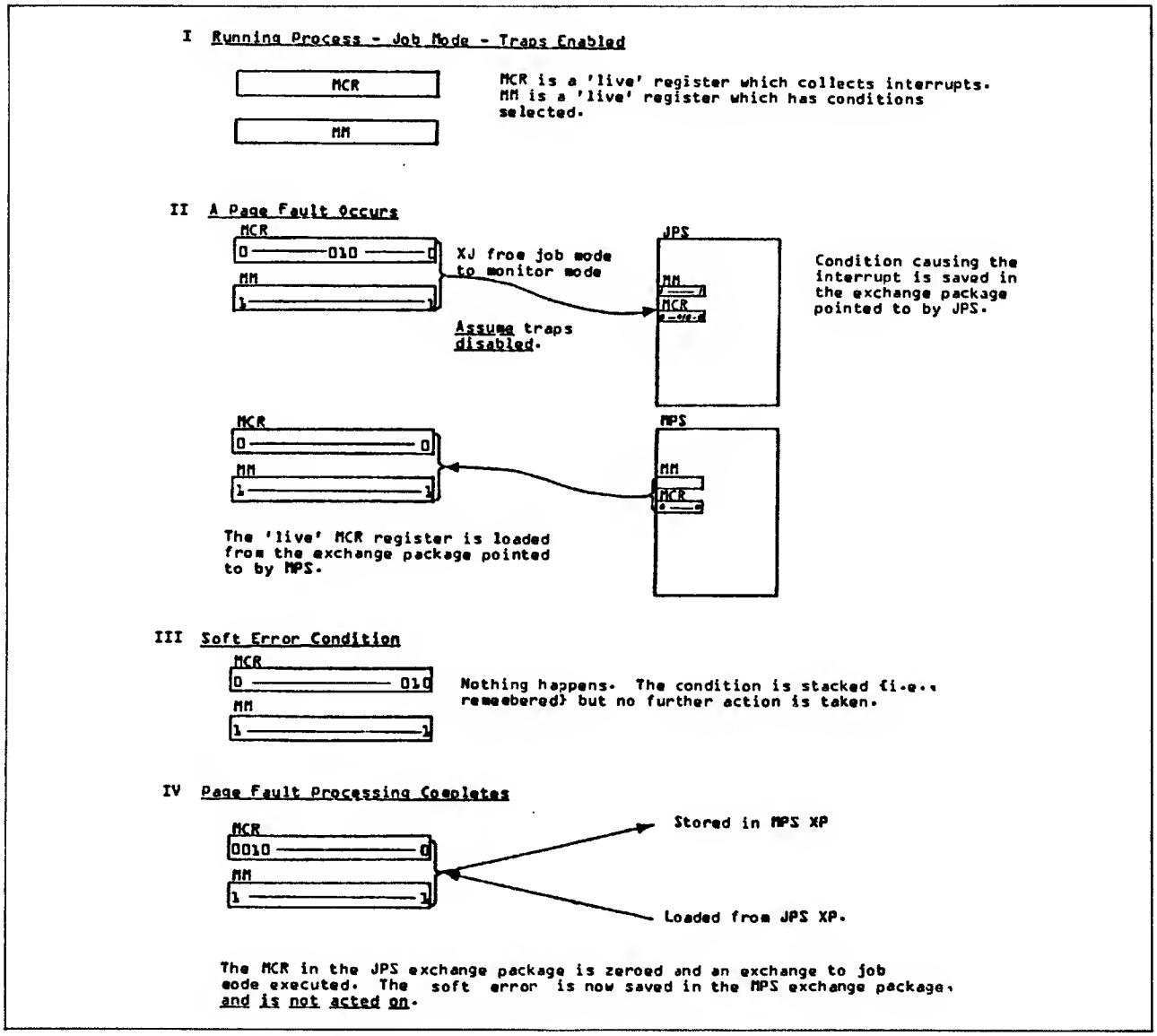


Figure 6-3. Examples of Interrupts

However, it is not necessary to have traps enabled for all monitor mode processing. The preferable sequence is to enter monitor with traps enabled, immediately disable traps, complete processing of the interrupt, enable traps, and return to job mode. Any conditions that have arisen during the interrupt processing are handled via an appropriate trap handler.

The interrupt system is hierarchical. The hierarchy does have a meaning and should be used. For conditions logged in the monitor condition register, the hierarchy is:

```
                +--> STACK
              --+
EXCHANGE -> TRAP
              --+
                +--> HALT
```

Thus, an interrupt occurring in Virtual State job mode causes an exchange to Virtual State monitor. An interrupt in Virtual State monitor with traps enabled causes a trap. An interrupt in Virtual State monitor with traps disabled causes either a stack or halt, depending on the specific interrupt. The system must spend as little time as possible processing interrupts with traps disabled, because a higher priority interrupt may be pending. Some care is necessary in this area when designing the operating system.

The interrupt processing, as it affects the MCR, is very similar for the UCR. This register collects user conditions that typically lead to a trap interrupt. These conditions are best handled from within the user's address space, built by a system routine. The hierarchy for these conditions is:

```
TRAP -> STACK
```

Thus, an interrupt with traps enabled causes a trap whether in a Virtual State job or monitor. An interrupt with traps disabled is stacked.

In other words, the condition may be acted on or remembered. However, interrupt handlers are organized so these conditions cannot arise, therefore stacking does not occur very often. As has been previously stated, the relationship between the UCR and the UM is the same as that between the MCR and MM. If a particular condition has not been selected by the user in the UM, then it effectively is stacked indefinitely. Certain instructions, such as floating-point arithmetic, yield results that could differ depending on the settings in the user mask. This occurs when end-cases such as exponent overflow and underflow are encountered. Also held in the UCR are four monitor conditions. The hierarchy for these is:

```
TRAP -> EXCHANGE -> HALT
```

Thus, an interrupt with traps enabled causes a trap whether in a Virtual State job or monitor. An interrupt with traps disabled causes an exchange to Virtual State monitor when an interrupt occurs in Virtual State job mode, and a halt when an interrupt occurs in Virtual State monitor mode.

The exchange and halt conditions should normally arise very infrequently or not at all, since the interrupt handlers can be organized to prevent them. These monitor conditions have been placed in the UCR for specific reasons. For example, a trap on an unimplemented instruction is intended to be used for a software simulation of an instruction that is not in the repertoire of Virtual State. This simulation must take place from within the user's address space. Other monitor conditions in the UCR have to wait until the system instructions have been discussed, in particular call/return.

A more detailed discussion of the interrupt system, where each condition is considered, is postponed until the stack processing characteristics of Virtual State are described. Some final points help to clarify the general process at this stage.

- The overall scheme of events is represented in figure 6-4. In this flowchart, stacked conditions lead to an RNI (read next instruction). As indicated in the previous paragraph, this is a conceptual process only.

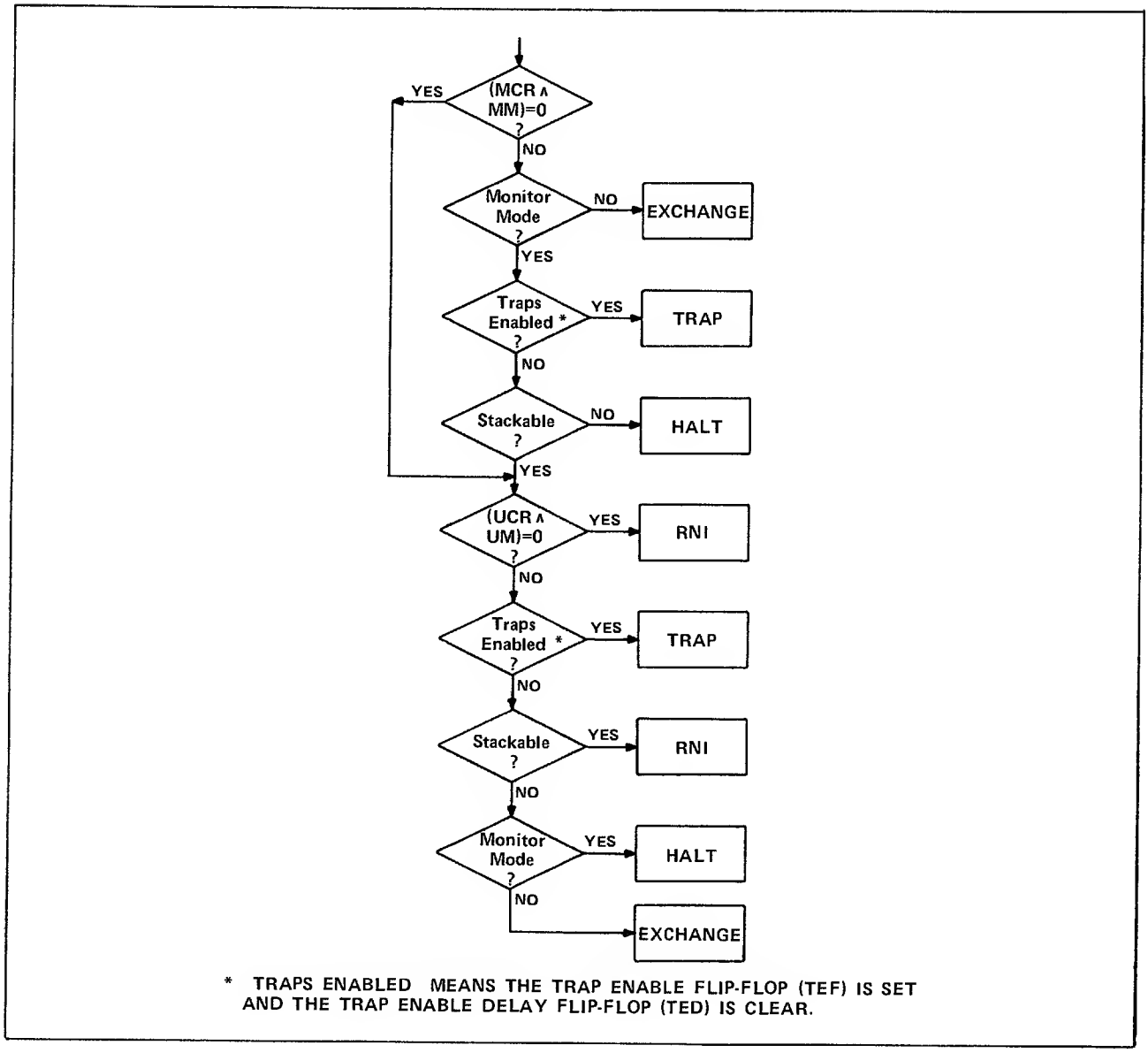


Figure 6-4. Interrupt Flowchart

- Conceptually, the hardware checks for interrupts before, during, and after instructions. In actuality, only uncorrectable errors can occur at any point, and the wrapup after one instruction and the prevalidation for the next can become essentially a single process.

- The hardware typically collects interrupts, not between instructions, but between the instructions' points of no return. There comes a point in every Virtual State instruction when something is written (memory, register file, and so forth). Once this happens, the instruction is committed, and, with the exception of hardware faults, interrupt conditions that arise apply up to the next point of no return.

The concept of a point of no return is important, since hardware errors that occur before this point can be retried. If the retry is successful, a soft error condition is recorded. Otherwise, a detected uncorrectable error (DUE) is flagged. The processor not damaged (PND) works in conjunction with the DUE feature. If a fatal error is detected before the point of no return, the processor sets the PND flag, indicating to the monitor that, although the processor is broken, the process is still intact. When the error arrives after the point of no return, but before instruction completion, the process is an unknown state and the PND is left clear.

The Virtual State call/return mechanism is the technique used to cross protection boundaries within an address space. It is also used to transfer control between procedures (subroutines). It is designed to satisfy the requirements of block-structured languages, permitting recursive calls such as CYBIL, the implementation language for Virtual State.

SOFTWARE CONSIDERATIONS

Before describing the call/return mechanism, a short introduction to block-structured languages is needed. Procedures (subroutines) in a block-structured language are organized into a series of nested blocks (figure 7-1). In each set of blocks, variables are related. Variables are classified into two types, static and dynamic. Static variables are allocated to fixed memory addresses and tend to be used throughout a program. Dynamic variables are allocated to a different memory address each time a procedure is called. This allocation occurs in a stack. A stack is an area of memory that can grow and shrink dynamically, in accordance with the demands. Each time a procedure is called, a new stack frame for that procedure is created. On Virtual State, much of the management of this stack is accomplished by the hardware of the call/return mechanism. The objective is to contain the code for a given function in a compartment for which there are controlled modes of entry. The variables used by this compartment (or block) are generated each time the block is entered, and are erased when the block is exited.

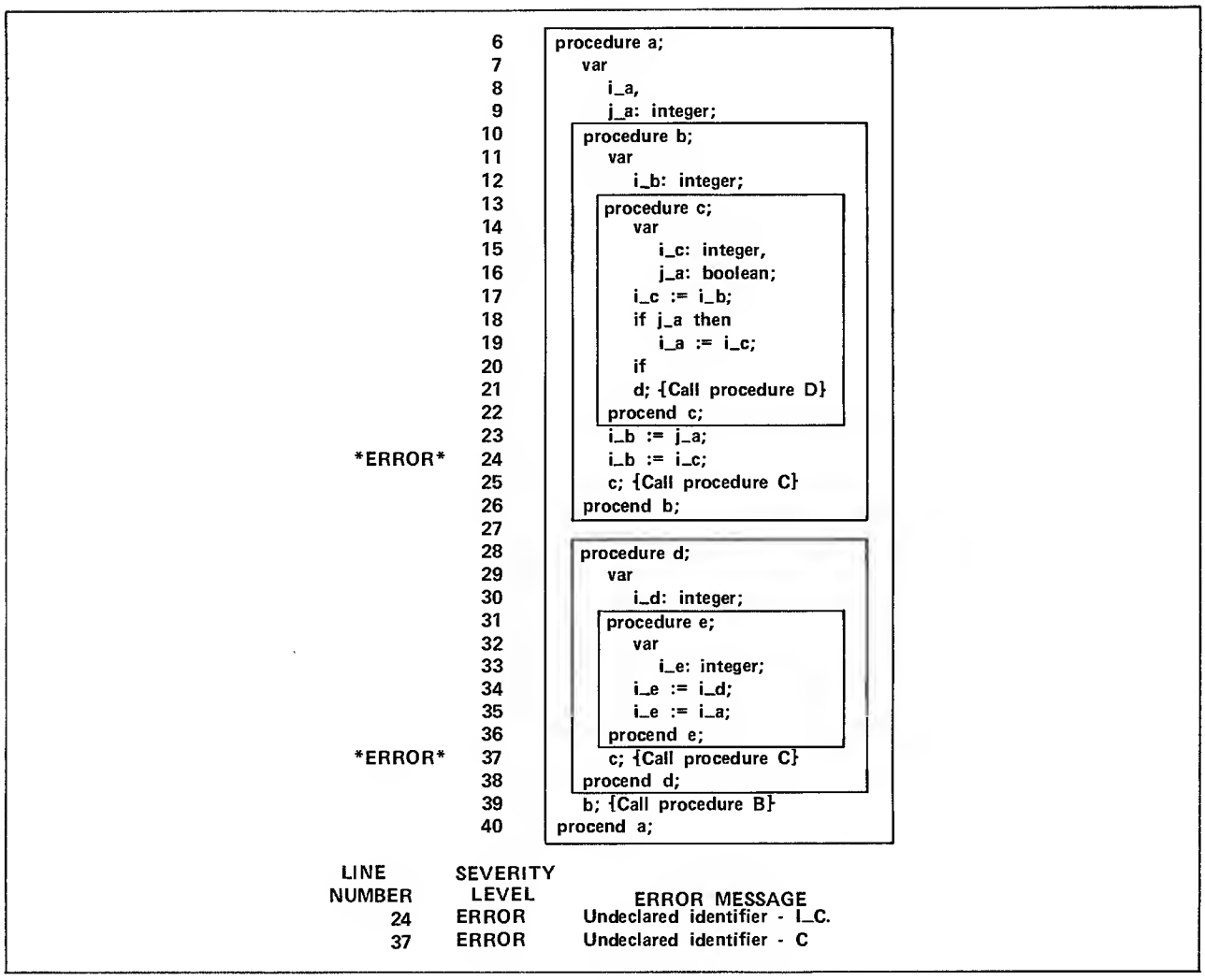


Figure 7-1. Example of Block Structure

In the diagram, two sets of nested blocks are shown in module A. These are B,C and D,E. The replacement statements in procedure C involve variables described in program module A and in procedures B and C. Knowledge of the whereabouts of these variables is maintained by a static link held in the stack frame for each procedure. This linkage is called static since it is known by the compiler at compile time and never changes.

Figure 7-2 illustrates the stack mechanism. The process starts by creating a stack frame for the dynamic variables in the procedure A. A current stack frame pointer (CSF) points to the beginning of this stack frame and a dynamic space pointer (DSP) points to the next available (free) space in the stack. On Virtual State, the stack frame and the pointers are established by software. When procedure B is called, procedure A's environment is saved and a stack frame created for procedure B. A dynamic link is created pointing to procedure A's stack frame, and a static link pointing, in this case, to the same stack frame. The dynamic link is termed the previous save area pointer (PSA) and is automatically updated on a call and return by the Virtual State hardware.

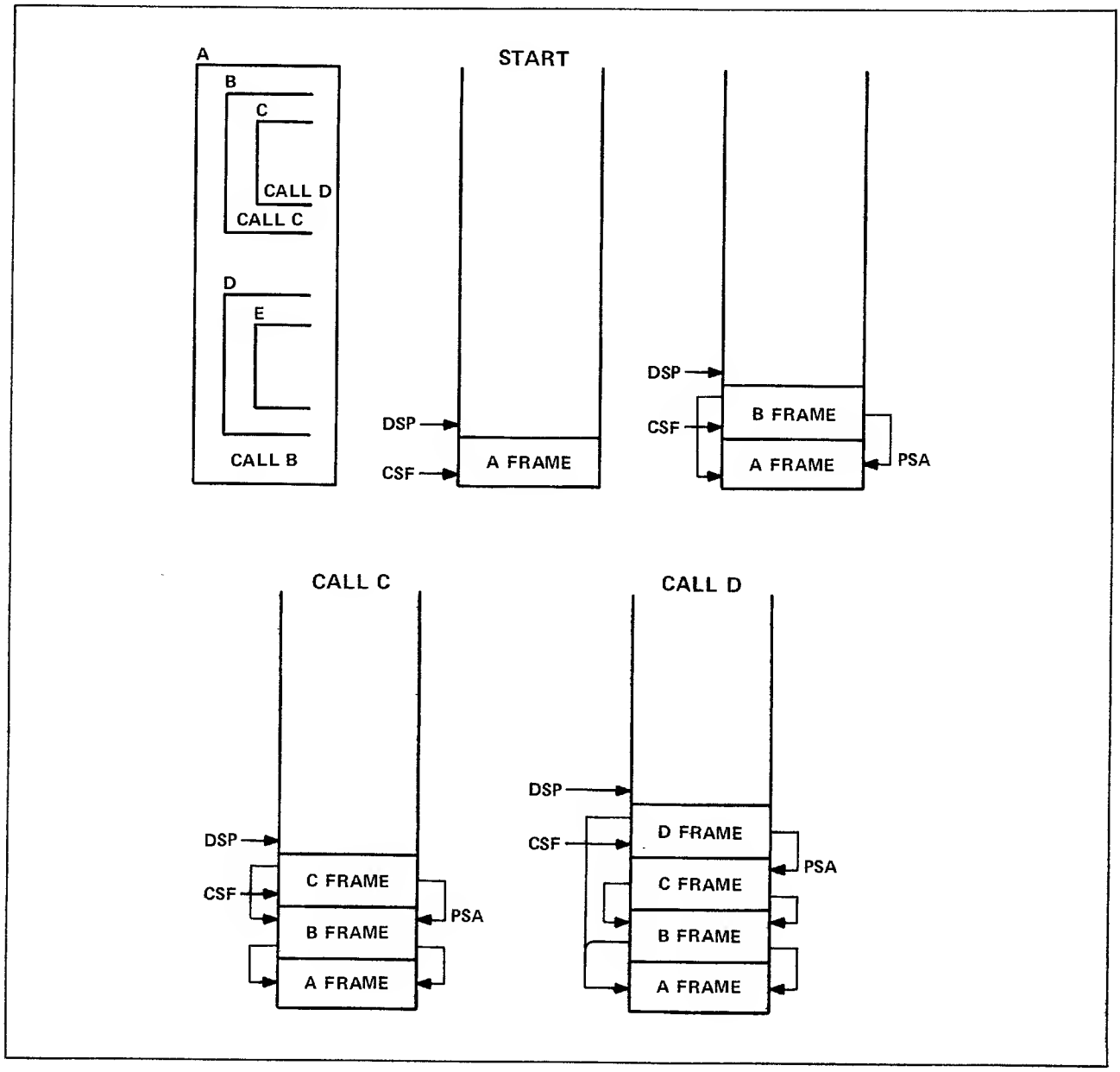


Figure 7-2. Stack Frame Manipulation by Call/Return

A call on procedure C follows in much the same way, and again the static and dynamic links point to the previous stack frame. When procedure C calls on procedure D, the dynamic link (for stack management) points to the previous stack frame. The static link, however, points to the stack frame for module A, but not to those declared in blocks B and C, which are contained within A but do not contain D. This occurs because procedure D is a block within base module A, and procedure D has access to variables declared in module A, but not to those declared in blocks B or C.

On each procedure call the DSP is updated to point to the next available space within the stack. This is a software function on Virtual State, and represents the reservation of an area in the stack large enough to accommodate all of the dynamic variables for a given procedure, a quantity known only to the software.

Since each time a procedure is called the caller's environment is saved, a procedure may be reentered or called recursively. This is true, providing all code is organized into pure procedures. No code modification is permitted.

The call/return mechanism provides facilities for protection, dynamic linking, and virtual machine switching. These features of call and return are developed separately because of their importance. First, it is necessary to understand the hardware support for the basic mechanism.

Consider an executing procedure, procedure A, that calls a second procedure, procedure B (figure 7-3). Four parameters are of interest:

- Top of stack (TOS) pointer in exchange package.
- Dynamic space pointer (DSP) held in A0.
- Current stack frame (CSF) held in A1.
- Previous stack area (PSA) held in A2.

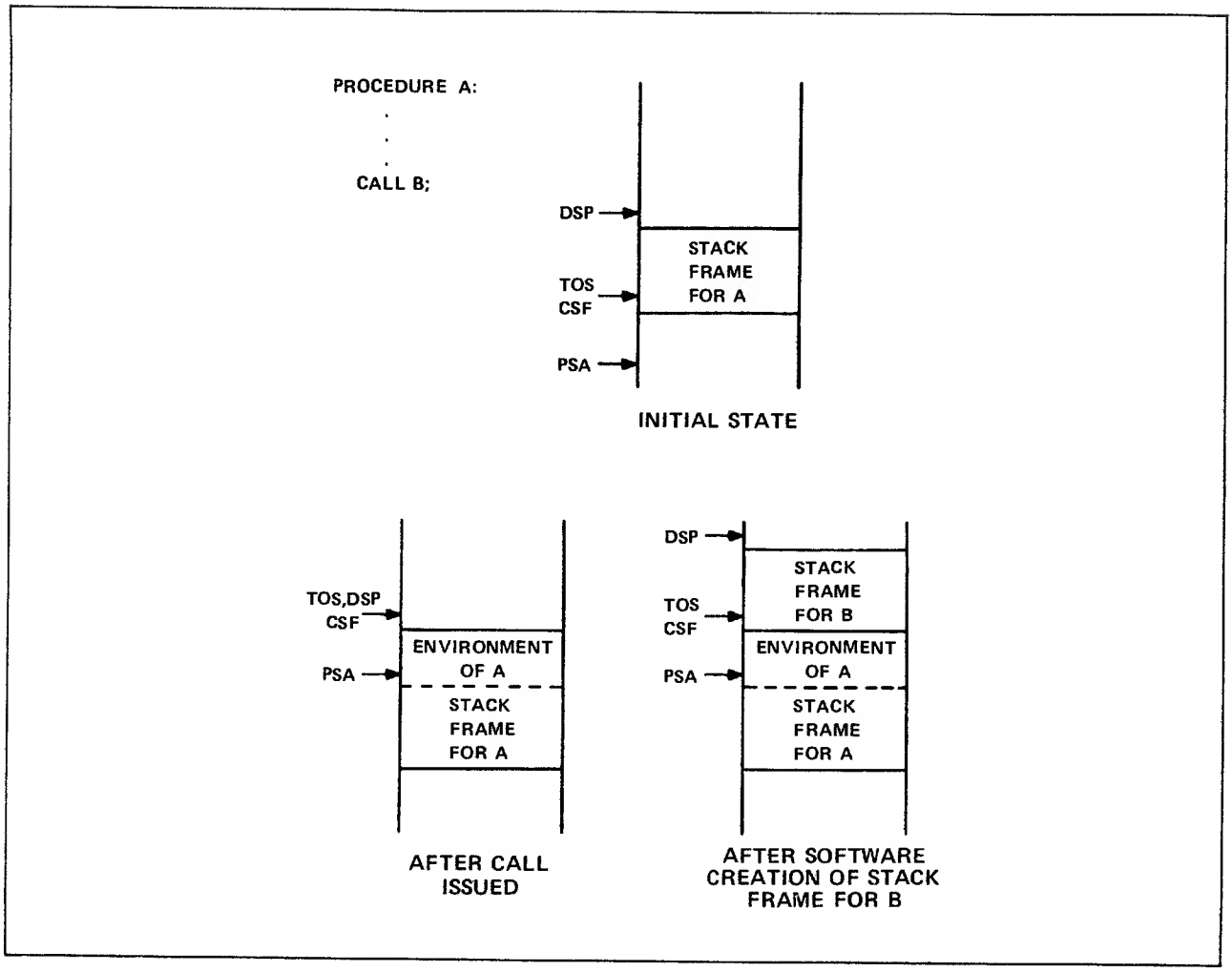


Figure 7-3. Basic Call Mechanism

These quantities are pointing within the stack as indicated prior to the call. When the CALL is issued, the following steps occur:

1. The caller's environment is saved in the caller's stack frame, and TOS is updated to reflect the next free space in the stack.

2. PSA is set to DSP.

3. CSF and DSP are set to TOS.

Next, the software creates a stack frame to hold dynamic variables for procedure B. At this time, the four key parameters are pointing into the stack for procedure B precisely as they had been for procedure A. Return can be accomplished easily, since the pointers to A's stack frame have been saved (in A0, A1, and A2) in the stack frame save area (SFSA) (figure 7-4).

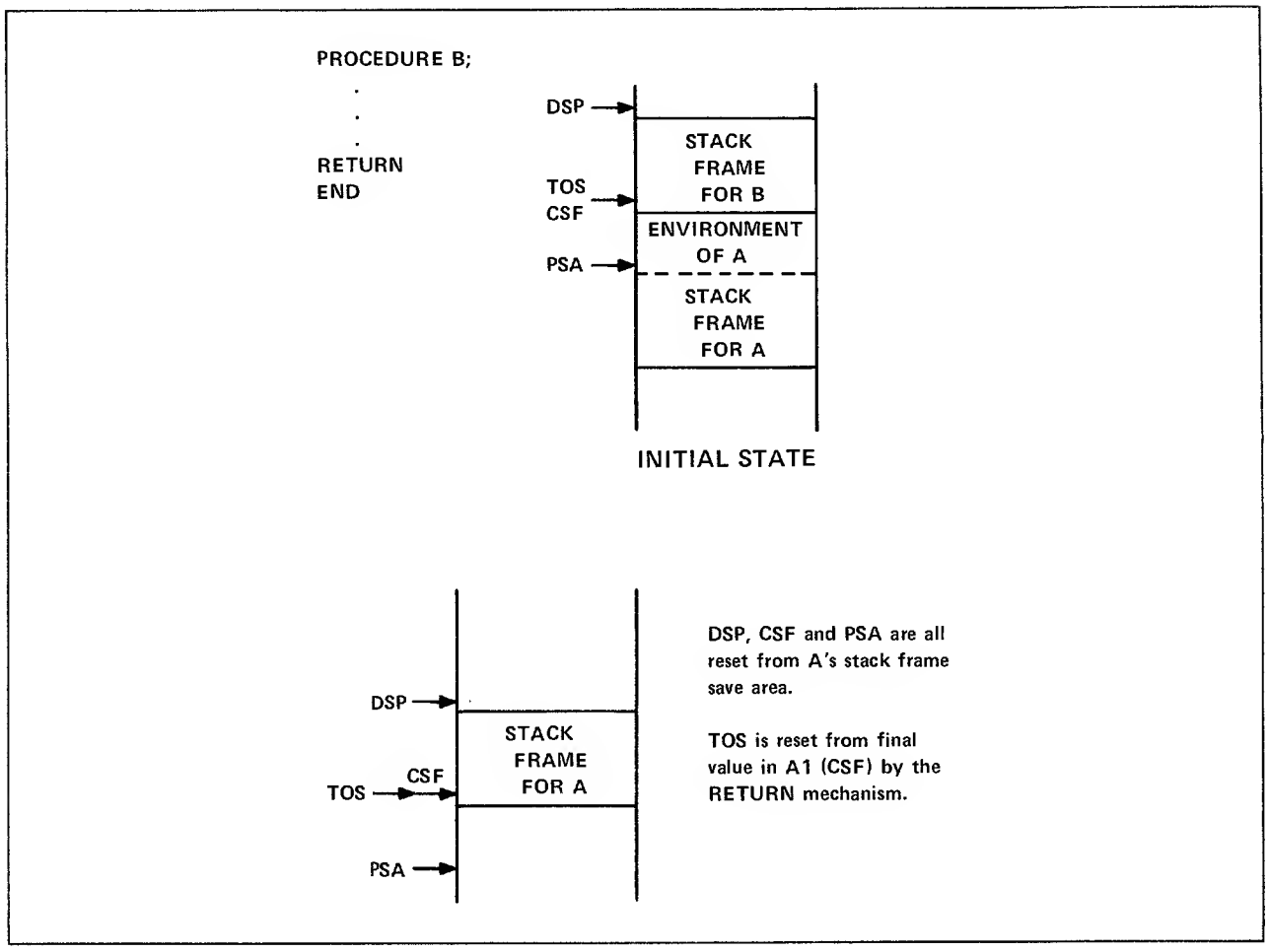


Figure 7-4. Basic Return Mechanism

CALL - THE BASIC MECHANISM

A stack is created by the operating system for each ring of execution. A TOS pointer for each of these stacks is kept in the exchange package. Whenever a procedure calls another procedure, the caller's environment is saved in the SFSA (figure 7-5). The first four words of this area are stored unconditionally, and the remaining words are stored under the control of the caller. The caller formats a stack frame descriptor in X0-right prior to issuing the call. The descriptor specifies which X and A registers are to be saved, in addition to those saved by default. Registers saved must be consecutively numbered. In the case of A registers, since A0, A1, and A2 are saved unconditionally, it is only necessary to specify the upper limit of the consecutive list. The descriptor is analogous to that used by load/store multiple instructions, which are described later. It must be supplied, and the terminal A register designator must be greater than or equal to 2. If no X registers are to be saved, then the terminal X register designator (Xt) should be less than the starting X register designator (Xs). When the callee returns to the caller, these registers are automatically restored. The operation of the hardware strongly suggests a software calling convention whereby the caller saves the environment.

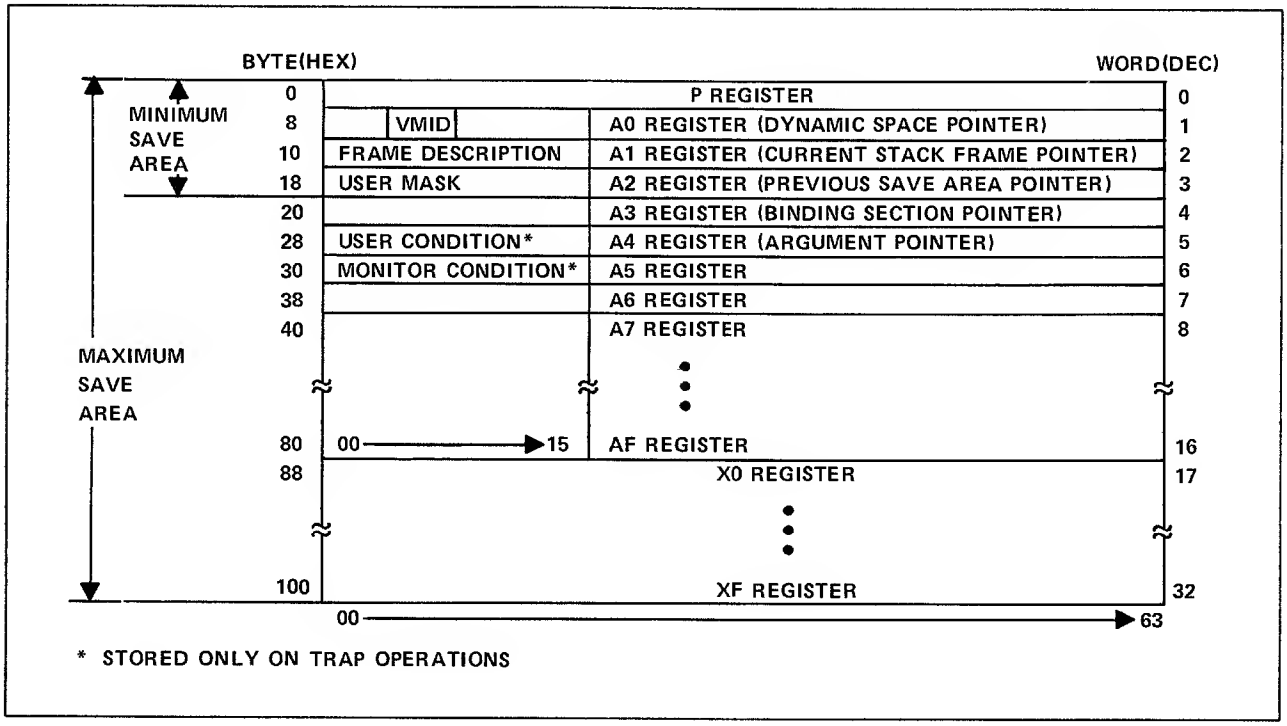


Figure 7-5. Stack Frame Save Area

Virtual State supports two forms of the call instruction that may be loosely regarded as general purpose (call-indirect) and special purpose (call-relative) calls. The call-indirect may call into a different segment in a different ring and, as will be shown later, into a different virtual machine. The call-relative, however, calls into the same environment. Although the same basic mechanism applies to both forms of the call, the general purpose version must guarantee the privacy of the callee and the caller, who may have quite different privileges.

The flowcharts at the end of this section describe these instructions completely, but the basic steps are:

1. The caller's environment is saved.
2. The caller's stack frame is pushed.
3. The P register is updated to point to the first instruction of the callee to be executed.

A single return instruction inverts this process.

1. The callee's stack frame is popped.
2. The caller's environment is restored.
3. The P register is updated (from the caller's environment), so it points to the first instruction to be executed following the original call.

General Notes

- The caller's environment is saved in the caller's stack. In fact, it is saved at the top of the caller's stack. To minimize the execution time for a call, this environment is stored on word boundaries. If the top of stack happens not to be on a word boundary, the stack frame save area is forced to a word boundary by the call instruction.
- The callee's stack frame is not automatically created. The CSF pointer is updated to point to the first entry in the stack frame, but it is the responsibility of the callee, via software, to reserve the appropriate amount of space in the stack. It is also recommended that an integral number of words be reserved for that purpose.
- Since the call relative calls to a word boundary, every procedure (subroutine) must start on a word boundary. This is not strictly necessary for external procedures. However, when the process of binding is described, the reason for this convention will become apparent.

RETURN - THE BASIC MECHANISM

The basic return mechanism pops the callee's stack frame and restores the caller's stack frame as the active frame. The environment that exists following the execution of a return instruction is precisely the environment that existed prior to the execution of the associated call instruction. Figure 7-6 illustrates the changes that occur in the stack when the sequence followed is:

1. Call (intraring).
2. Call (inter-ring from ring 11 to ring 3).
3. Return.
4. Return.

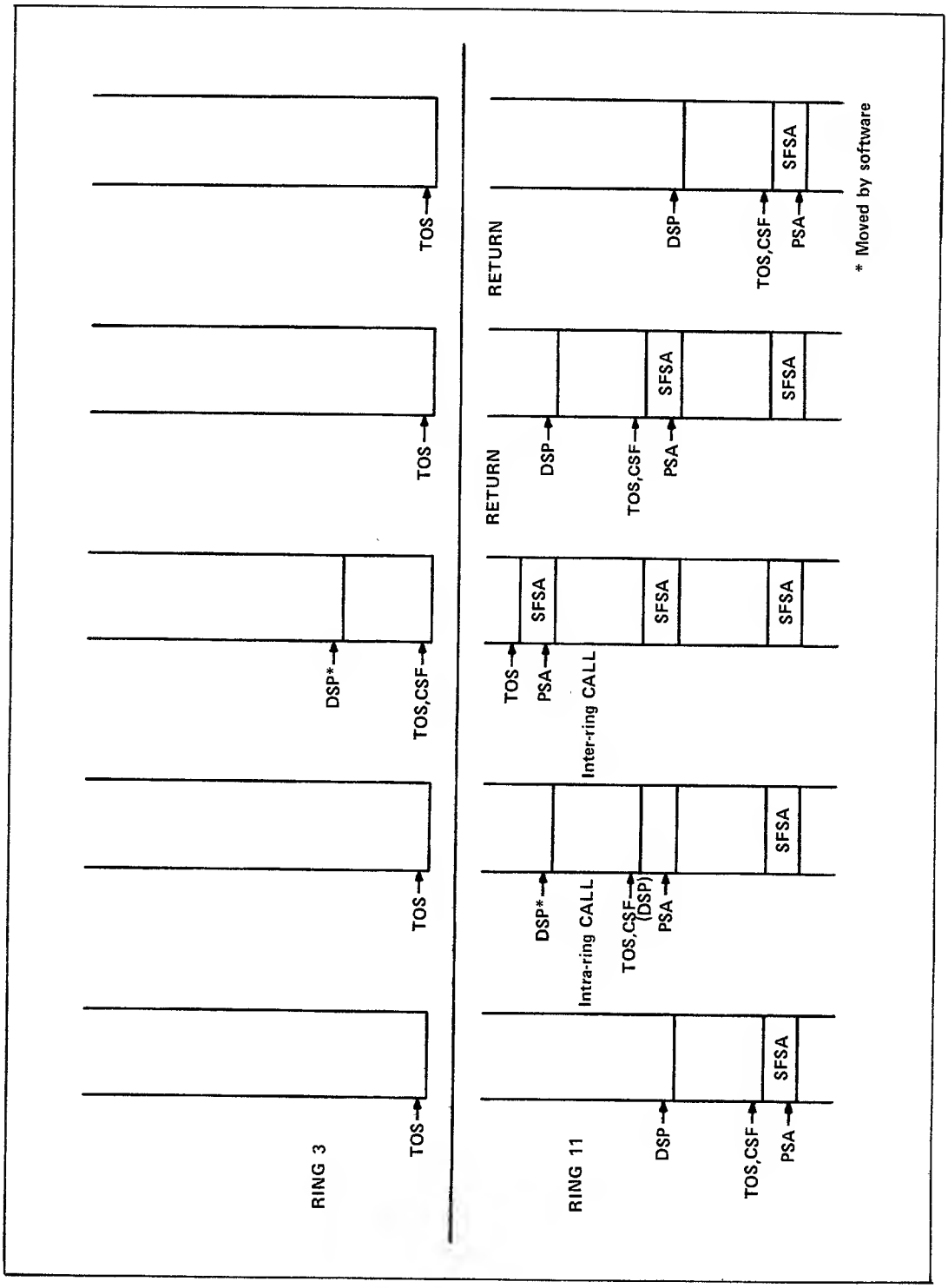


Figure 7-6. Call/Return

Since calls are typically to inner rings, returns are typically to outer, less privileged rings. It must be ensured that the callee's greater privileges are not transmitted to the caller. The callee's ring number may appear in any A register used by the callee, not just those saved by the caller. To ensure that this ring number does not get returned to the caller, the return instruction checks that no A register is returned to the caller with a ring number that exceeds the caller's ring number. This process is termed rippling (figure 7-7). The caller's overall privileges, maintained in the P register, are automatically restored when the caller's P register is loaded from the SFSA. A check is made to ensure that the keys loaded are identical to those found in the caller's SDE.

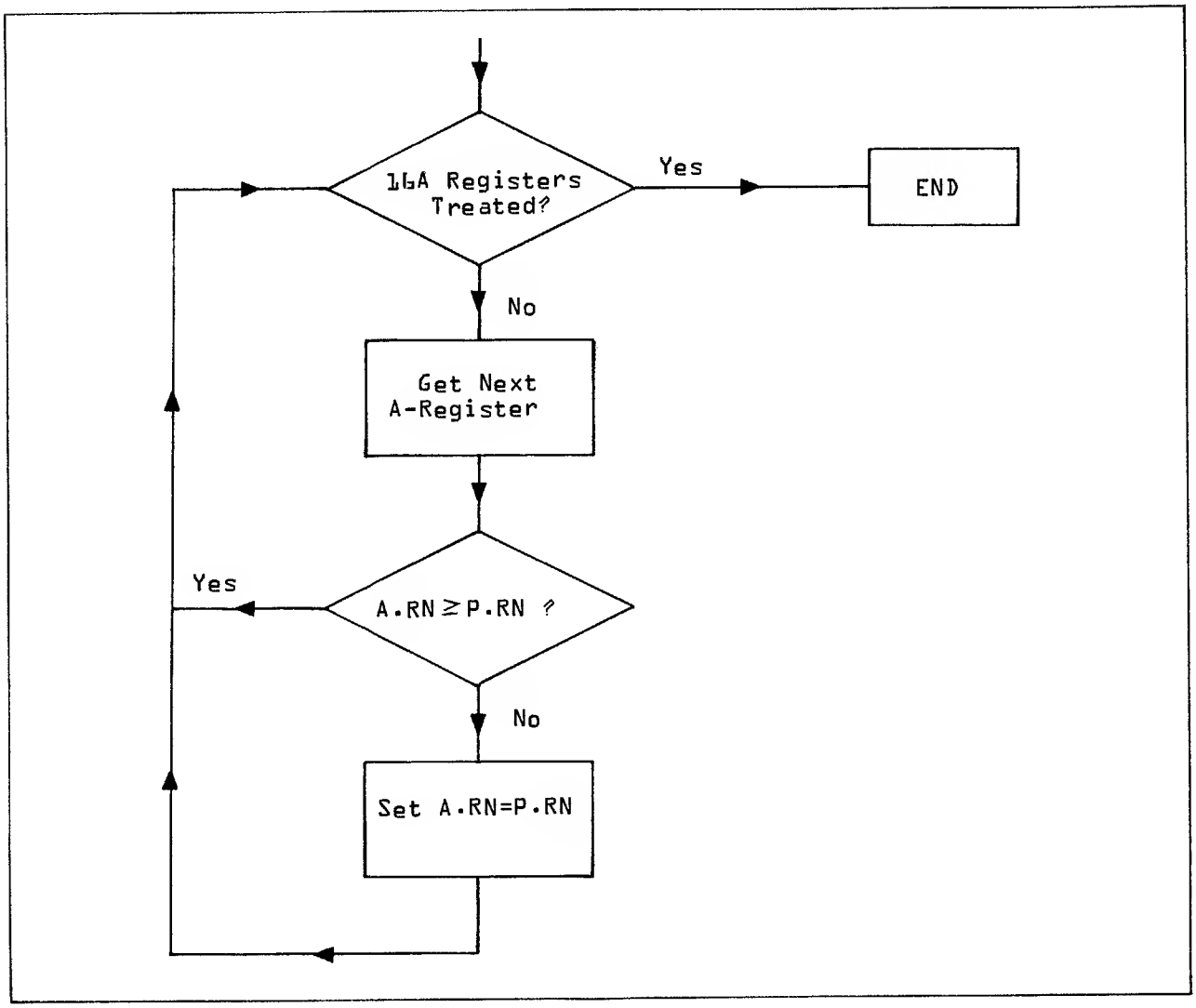


Figure 7-7. Rippling

General Notes

- Processes typically start execution in their outermost ring. Stacks in all rings are empty, except for the one in the primary ring of execution. As calls are made inward, entries are made in other stacks that are emptied as returns are issued. The question might be asked, Why have fifteen stacks? Again, when the security of the system is considered, the reason becomes obvious. Since the stack holds the dynamic variables for an executing process, that process has read/write access to the stack. If there is only a single stack, an executing process could make a call to a procedure in an inner ring and access that procedure's dynamic variables at the top of the stack. The only way to prevent this is for the callee to zero out all dynamic variables used. This is prohibitively time consuming.

- Call and return are time-consuming operations and are designed to satisfy the general architectural requirements of Virtual State. In particular, the generalized form of call (call indirect) should only be used when an external procedure call is made to a procedure in another segment. When binding is discribed, it will be seen that the binder assists with this task.

The flowcharts at the end of this section describe the overall process for return.

POP - THE BASIC MECHANISM

There are times, typically in the presence of an error or a nonlocal GOTO, when an entry or a number of entries must be eliminated from a particular stack. Since these entries have been created by a series of calls, a similar series of returns accomplish the required purge. However, when the purging is to be completed without executing intervening instructions, this is only achieved by an appropriate software sequence, or by issuing a pop instruction provided for this purpose. The pop instruction simply moves the CSF, PSA, and TOS pointers, eliminating the stack frame but not changing the P-counter. Figure 7-8 is an example, wherein calls have been made three deep into the structure of a program, and the entire set of calls has been aborted. Pops can only be issued within the current ring of execution. Access violations are not checked, and if a pop is attempted across rings (as indicated by the ring number in A2-PSA), the instruction execution is inhibited and the program interrupted.

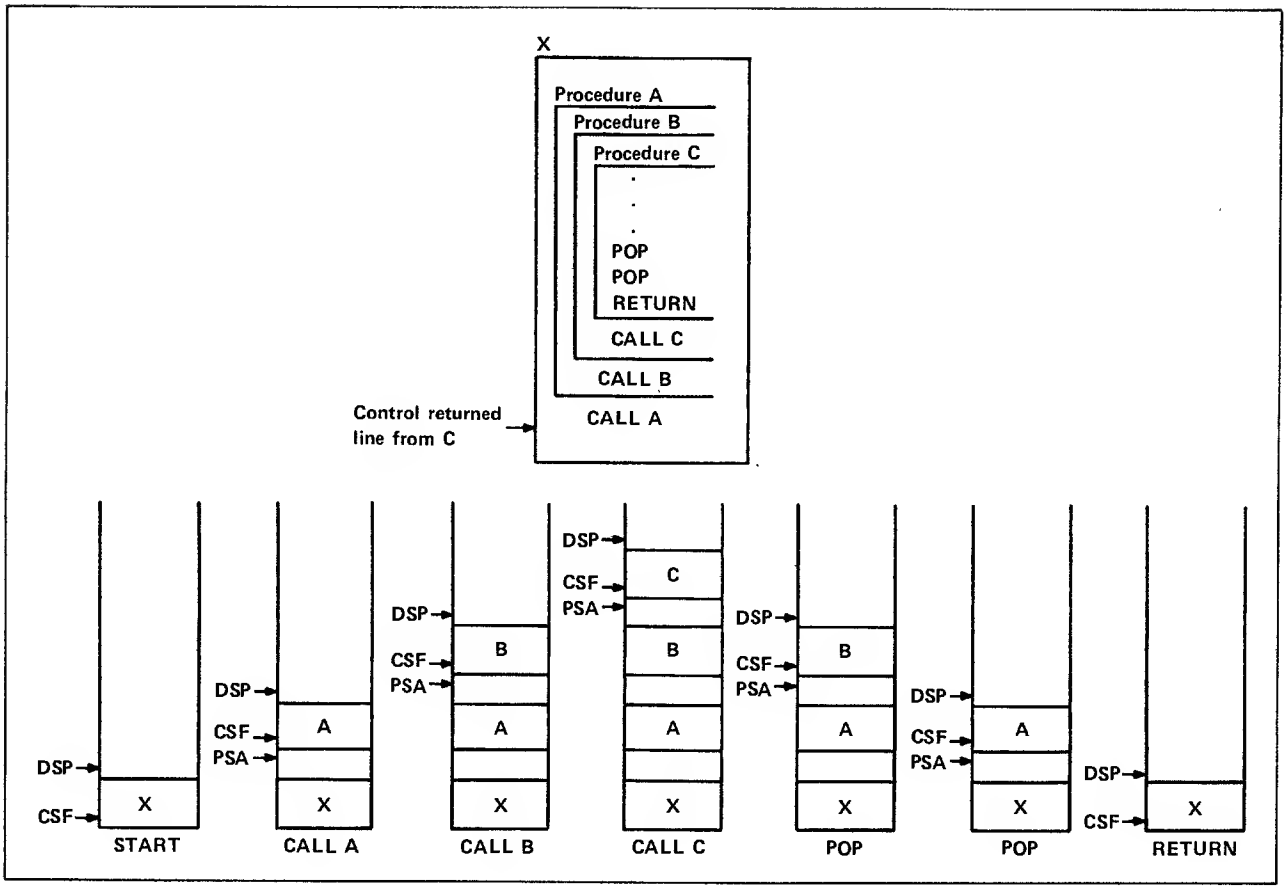


Figure 7-8. Example of Pop Instruction

THE BINDING SECTION - CODE SHARING

The entry to procedures must be carefully controlled. Procedures must receive control only at the points where they expect to receive control, that is, their entry points. To make this possible, procedures are not entered directly, but are entered via a pointer to the procedure. This pointer is held in a binding section. All such pointers are placed in the binding section by the loader, and the call mechanism guarantees that the call is made via a binding section.

The objective on Virtual State is to have one copy of a code segment located in memory and shared by several users. Each user has a copy of each code sequence required in his or her virtual memory address space. For example, the FORTRAN compiler exists only one time in real memory, but depending on the user who has the CPU, the compiler operates on different compilation units. There must be nothing in the code segment that makes a direct reference to modified data. This is accomplished by placing pointers to such data in a binding section created along with each code module, and then giving the address of the binding section to the callee when the procedure call is invoked.

Each executing task has some code that it may be sharing with other tasks, and some data typically unique to itself. When a compiler compiles some source code, it compiles offsets into the binding section and directives to the loader for building the binding section. It is then the responsibility of the loader to link all code modules and build the necessary binding sections. This process is described more fully in volume II of the Virtual State hardware reference manual (refer to the preface for more information).

The binding section is in a separate segment and is identified uniquely by its SDE (refer to the section on virtual memory). It typically contains pointers to external procedures and pointers to working storage areas that hold static variables. Static variables do not appear in a stack frame. When a procedure calls on another externally defined procedure the call points into the binding section. The binding section, by convention, has one or two full-word entries containing a code base pointer (CBP) and a pointer to the callee's binding section (figure 7-9). The CBP points to the first executable statement in procedure D. The VMID and R3 fields in the CBP are discussed shortly. The external procedure flag (EPF) field in the one state indicates that the procedure being called is an external procedure, and therefore the next entry in the binding section is the pointer to the callee's binding section. This field is nothing more than a flag to differentiate between single-word and double-word entries in the binding section.

Whenever a call is made to a procedure in another ring, this form of the call instruction (via the binding section) must be used. However, for critical (intrasegment, intraring) calls, a shorter form of the call instruction should be used. This form finds the first executable statement of the caller at P plus an offset and obviates the need for a CBP. The offset used by this instruction is a 16-bit long-word offset; therefore, all procedures must start on a word boundary.

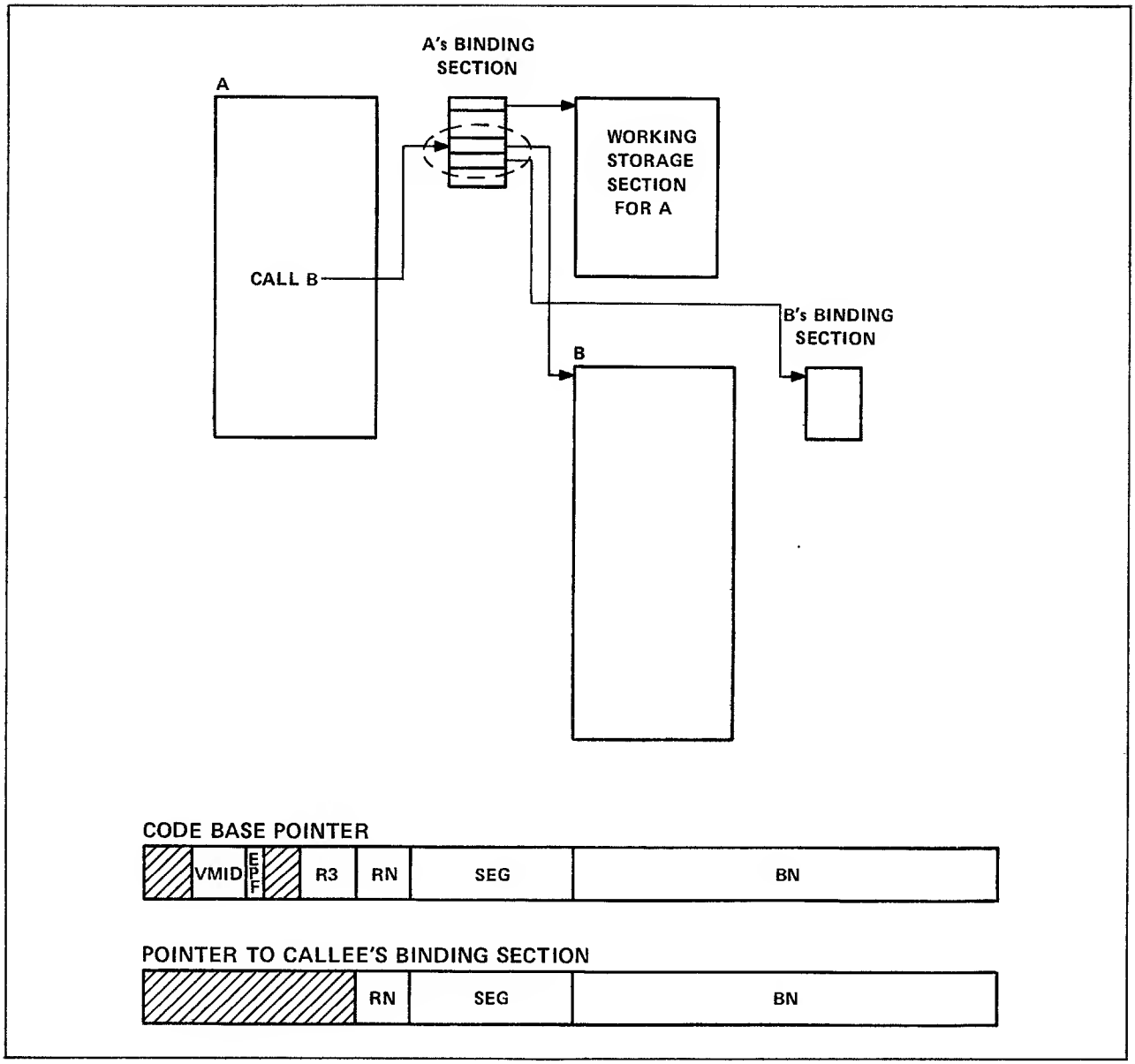


Figure 7-9. Call Indirect Example

To make code sharing possible, each task sharing the code must have its own data. The location of this data is defined through the binding section, defined via the process segment table. The segment table address (STA) is defined by the exchange package for that process. Each instance of a process has an exchange package that describes the process state registers for that exchange interval. This defines the whereabouts of the code and data to be used by the process, via the virtual memory mechanism. Different tasks using the same code have their own segment tables and have unique entries in the SPT for their binding sections and data. However, the page table entry for the code segment is shared by all tasks sharing the code (figure 7-10). This is quite simple if one remembers the basic virtual memory address translation mechanism.

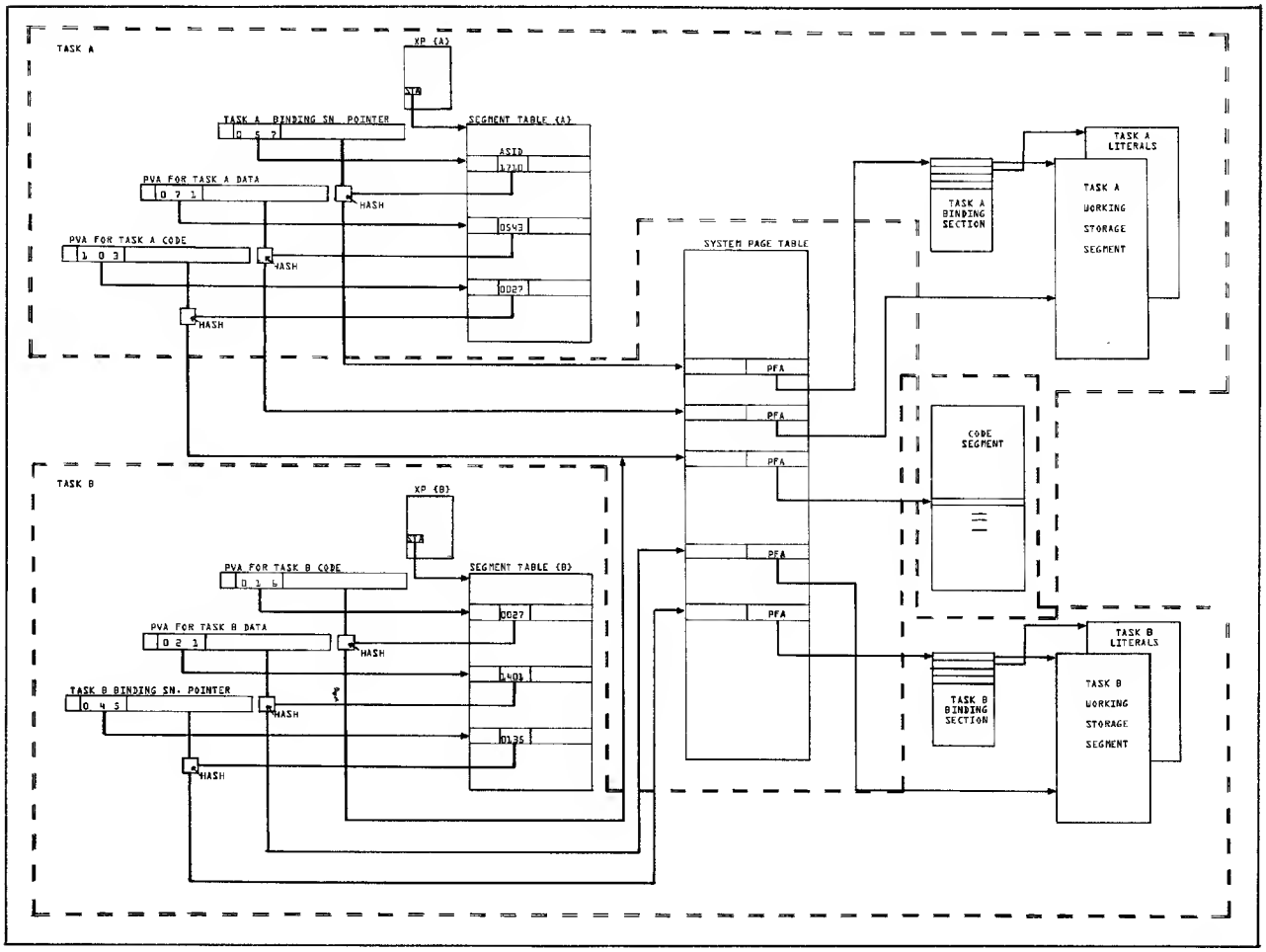


Figure 7-10. Code Sharing

The key to code sharing lies with the concept of the SVA. Shared code actually resides as two conceptually separate copies in two address spaces. When this code is referenced via a PVA, the first step is to translate the PVA into an SVA. The operating system arranges for code to be shared to have common SVAs. The translation to a real memory address (RMA) results in the same locations in central memory, regardless of the process requesting the translation. This is accomplished by assigning common ASIDs to shared code segments, which happens the first time the segment is referenced. Neither the originator of the shared code, nor any users of it, need be aware that the code is being shared. The only contingency is that code be organized into pure procedures. Code sharing by itself is unrelated to call/return. However, the separation of code and data, the absence of direct references to the data in the code, and the binding section all play their part. When a procedure is called from another procedure, the caller gives the callee's binding section to him or her. That is, the caller carries a pointer to the callee's binding section as a parameter of the call. It is by this mechanism that shared code (that is, code shared in real memory) receives different data sets to work on. The whereabouts of these binding sections is determined by the loader, which loads multiple copies of the code to be shared into virtual memory (figure 7-11).

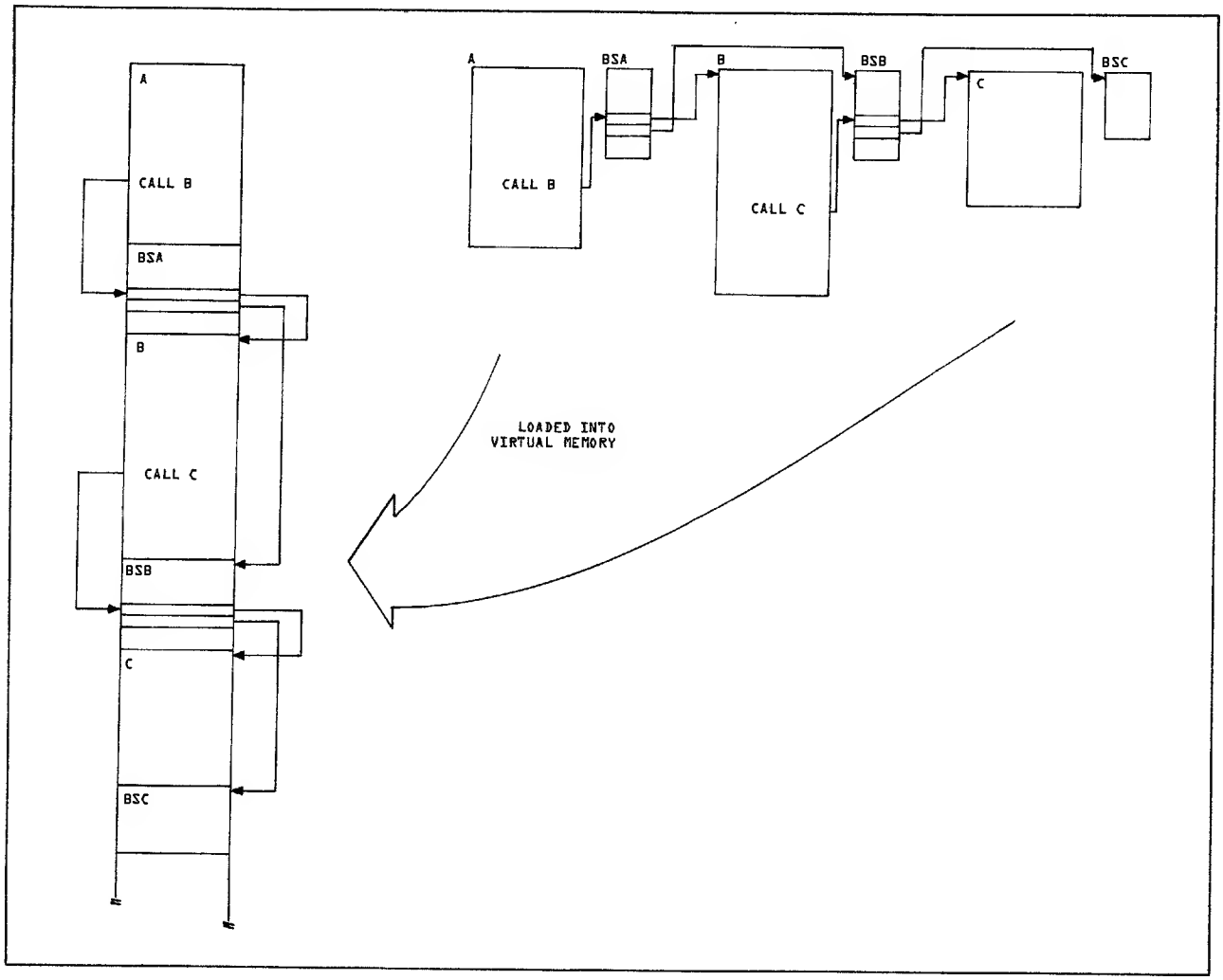


Figure 7-11. Loading Mechanism

Two additional areas, software conventions and parameter passing, need to be discussed before the basic mechanism can be summarized.

Parameter passing is discussed first. In general, when a procedure is called from another procedure, parameters need to be passed between them. The general parameter-passing technique selected for Virtual State is to pass an argument list pointer to the callee. Typically, this argument list pointer points to a list of pointers, which in turn point to data to be referenced by the called procedure. By convention, the argument pointer is held in A4, and the convention is supported by the hardware that transfers the argument list pointer to A4 during the execution of a call instruction.

Two other pointers are used by procedures, the binding section pointer and the static link. Of these, the binding section pointer is by far the more important and by convention is held in A3. As with the argument list pointer, this software convention is supported by the hardware. The choice of registers A3 and A4 to hold these quantities simplifies the saving and restoring of them during calls and returns, since the instructions always save a consecutive set of A registers, and A0 through A2 are always saved by a call. The static link is not always required, and for those cases where it is needed, it is carried by software, and the hardware has no part in its maintenance.

FLAGS

Two flags are handled by the call/return/pop instructions. These are the on-condition flag (OCF) and the critical frame flag (CFF). They are software flags reset by the hardware on each call to a new procedure.

ON-CONDITION FLAG

The end user causes the OCF to be set by requesting that a particular code sequence be executed when a chosen error arises. This is generally done via a high-level language, and the compiler generates the code necessary to set the OCF and generate a dummy stack frame for the on-condition processing (figure 7-12). A pointer in the user's stack frame points to this dummy. All exception conditions are typically selected by the process monitor. When one arises, a trap interrupt occurs, and the trap handler searches the stack for the presence of a set OCF. On conditions are set by a particular procedure. When a call is made, the OCF associated with the calling procedure is saved in the stack frame save area (SFSA) as part of the caller's environment, and the OCF is cleared. If an appropriate exception arises, it is handled by the caller's on-condition action, unless the callee had also requested specific action to be taken on the same exception. The following should be remembered.

- Actions to be taken on exceptions are specified by the user. They are recorded in a dummy stack frame by setting the OCF.
- Actions are established by a given procedure but carry across procedure calls.
- Each procedure may have its own unique set of on conditions.

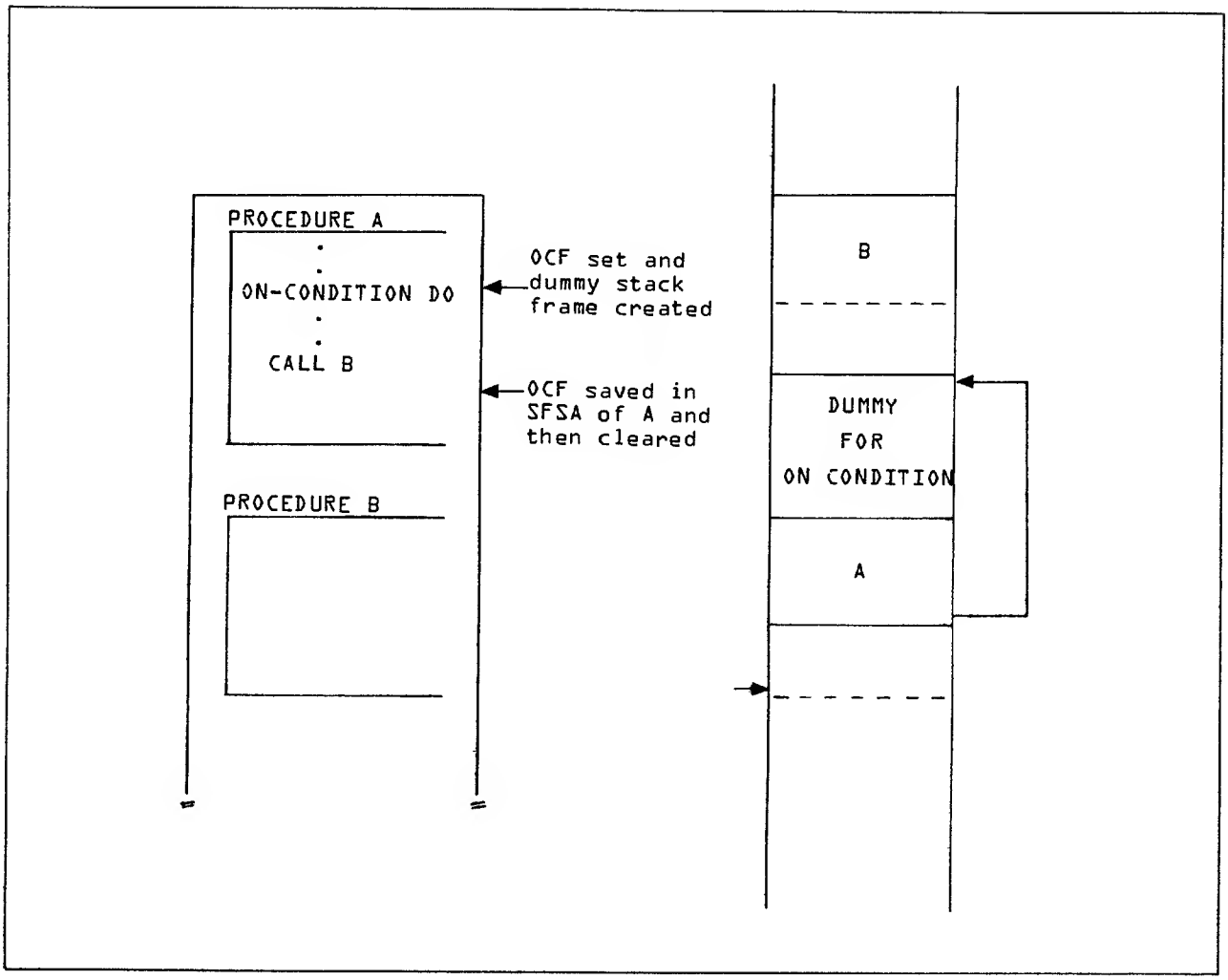


Figure 7-12. On-Condition Handling

CRITICAL FRAME FLAG

The critical frame flag (CFF) is a software device used to declare a procedure critical. The term critical is used to denote that some cleanup is required before leaving the procedure in question. In other words, exit from the procedure must take place in an orderly manner. An example helps to clarify this.

Imagine a job running under the control of a subsystem. The job may open a file or set some locks that must be either closed or cleared before the job is terminated. If the job terminates abnormally, the standard cleanup procedure pops the stack frames in use prior to returning to the subsystem for final exit. However, in the case where particular action is required before a stack frame is eliminated, a different path must be followed. The CFF is used to alert the subsystem in control of this situation. When the locks are set or the files opened, the CFF is also set, and subsequently saved in the procedure's SFSA, whenever it calls on another procedure. An attempt to pop a stack frame with the CFF set is detected by the hardware, and a trap interrupt is taken. The trap handler hands control back to the subsystem that, by an investigation of the user's stack, can perform the necessary cleanup operations.

OUTWARD CALLS/INWARD RETURNS

Calls can be made only within the same ring of protection or to an inner ring (figure 7-13). Hardware prohibits calls to an outer ring. However, when the operating system initiates (transfers control to) a user's job, a call must be made to the outer ring where the user's program resides. Since the hardware prohibits outward calls, the operating system software performs a return to the outer ring. Similarly, the "inward return" to the operating system is accomplished by using an inward call.

Only outward calls made by the operating system will be performed. Outward calls initiated by the user are not allowed. (Calls are used for transfer of control, not for transfer of data. Thus, a user may access an outer ring for data, since a call is not being performed.)

When an outward call is attempted, an interrupt occurs and the following steps are taken (assuming the machine is in the job mode with traps enabled).

1. An exchange interrupt occurs when the outward call is attempted.

2. The exchange interrupt handler sets the free-flag and issues an exchange. (The free-flag is a mechanism for converting an exchange interrupt into a trap interrupt.) This causes control to return to the original outward call instruction. However, before the original call can be retried:

    a. A trap interrupt occurs (because free-flag sets). This is an implicit call into the stack in the operating system's ring of execution.

    b. The trap handler creates two dummy frames in the callee's (user's) stack. The first dummy frame is the user's eventual stack frame; the second is created simply to be popped via a return that transfers control to the callee.

    c. The trap handler then executes a return to the callee. This pops the second dummy stack frame (figure 7-13).

Therefore, to perform an outward call, the interrupt handler software arranges to execute an outward return to the user.

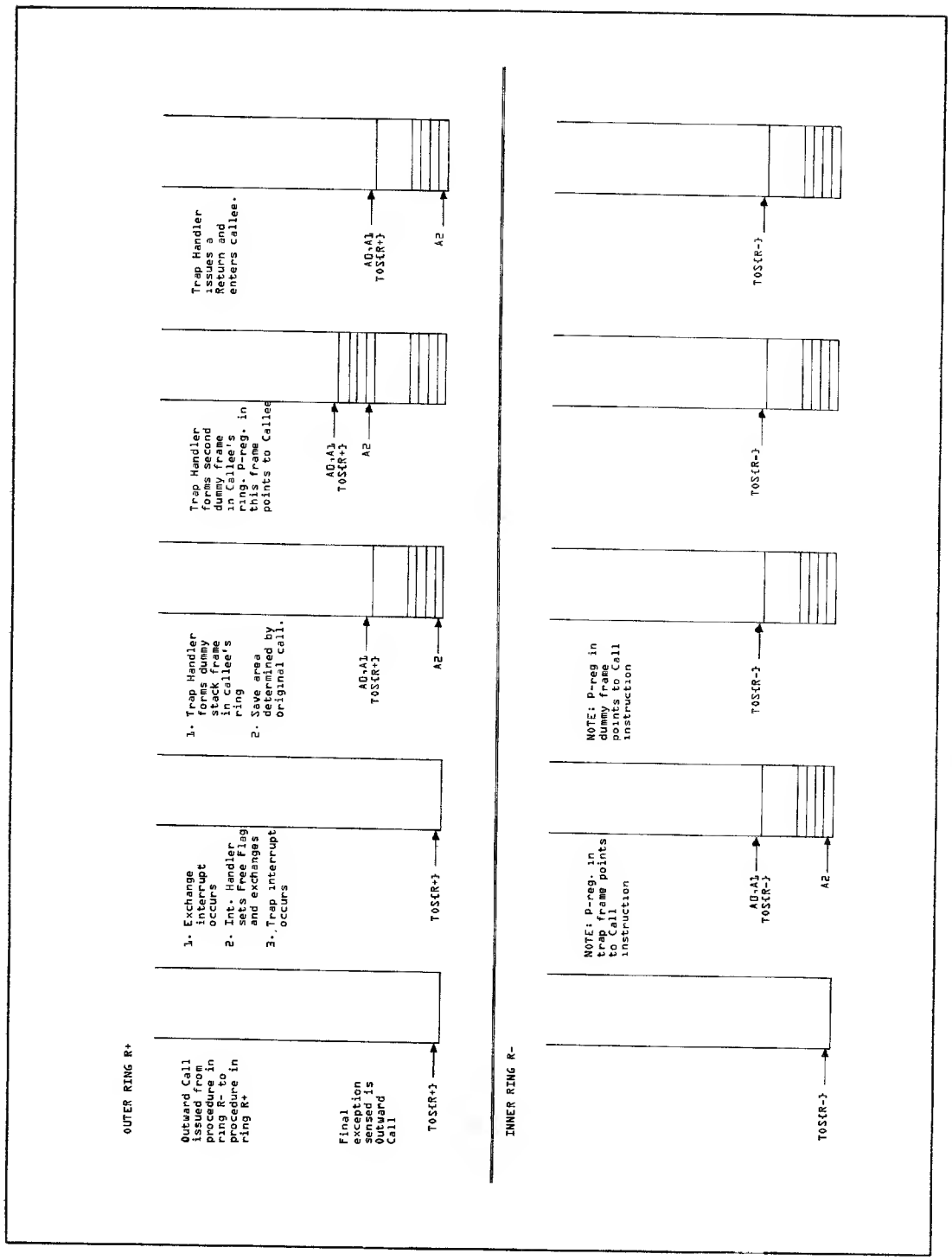


Figure 7-13. Outward Call

An inward return is the reverse of the outward call procedure and consists of executing an inward call back to the operating system, as follows (figure 7-14):

1. An exchange interrupt occurs (inward call).

2. The exchange interrupt handler sets the free-flag and issues an exchange. This causes control to return to the original inward return instruction. However, before the original return can be retried:

   a. A trap interrupt occurs (because free-flag sets). This is really an implied call into the stack in the callee's ring of execution.

   b. The trap handler calls on the task monitor in the operating system's ring of execution.

   c. The task monitor then:

      (1) Eliminates three frames from the callee's stack.

      (2) Adjusts its own stack frame to point to the operating system's stack frame.

      (3) Issues a return.

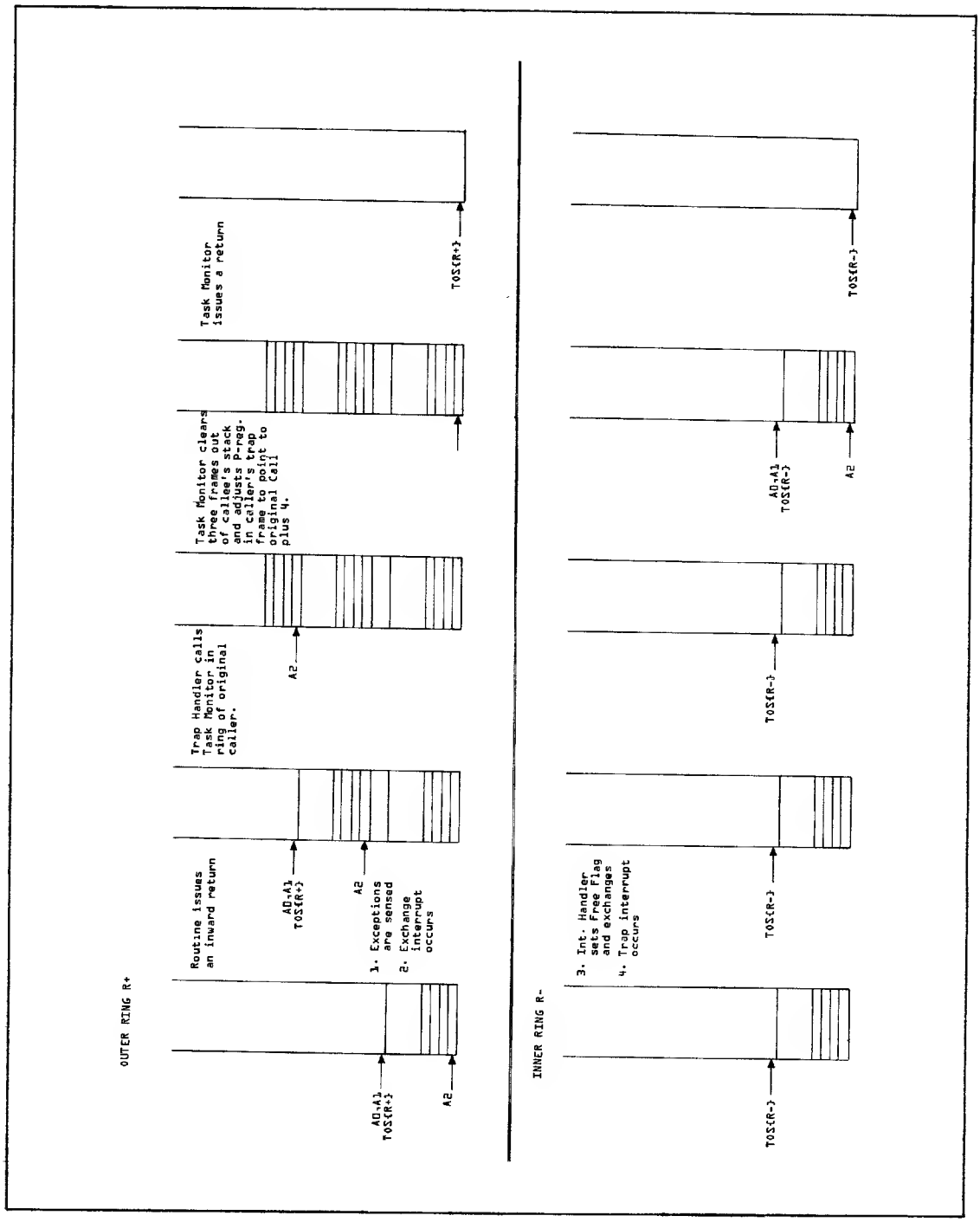


Figure 7-14. Inward Return

An outward call and inward return is a time-consuming process and should be used sparingly. A more efficient method of transferring control does exist, however. The operating system can determine in advance when an outward call is to be issued. If an outward call is detected, the operating system will not attempt to execute the call, and will not take the exchange jump to force a trap interrupt. Instead, the functions that would have been performed by the trap handler are performed by the operating system. Therefore, an outward return can be issued directly to transfer control to the callee.

When an outward call is made, the operating system assumes that there will be a subsequent inward return. Again, the functions that would have been performed by the task monitor if the prohibited inward return were issued are performed by the operating system. The operating system accomplishes this by calling on an outward-call service procedure that creates two stack frames in the callee's stack. It will seem as though the callee was called by a service procedure in his own ring of execution and that he called the original (outward-call) service procedure.

The outward-call service procedure then returns to the operating system to transfer control. The callee subsequently returns to an inward-return service procedure in his own ring of execution. This service procedure pops its own stack frame before making an inward call on the original outward-call service procedure. The outward-call procedure then returns control to the operating system (figure 7-15).

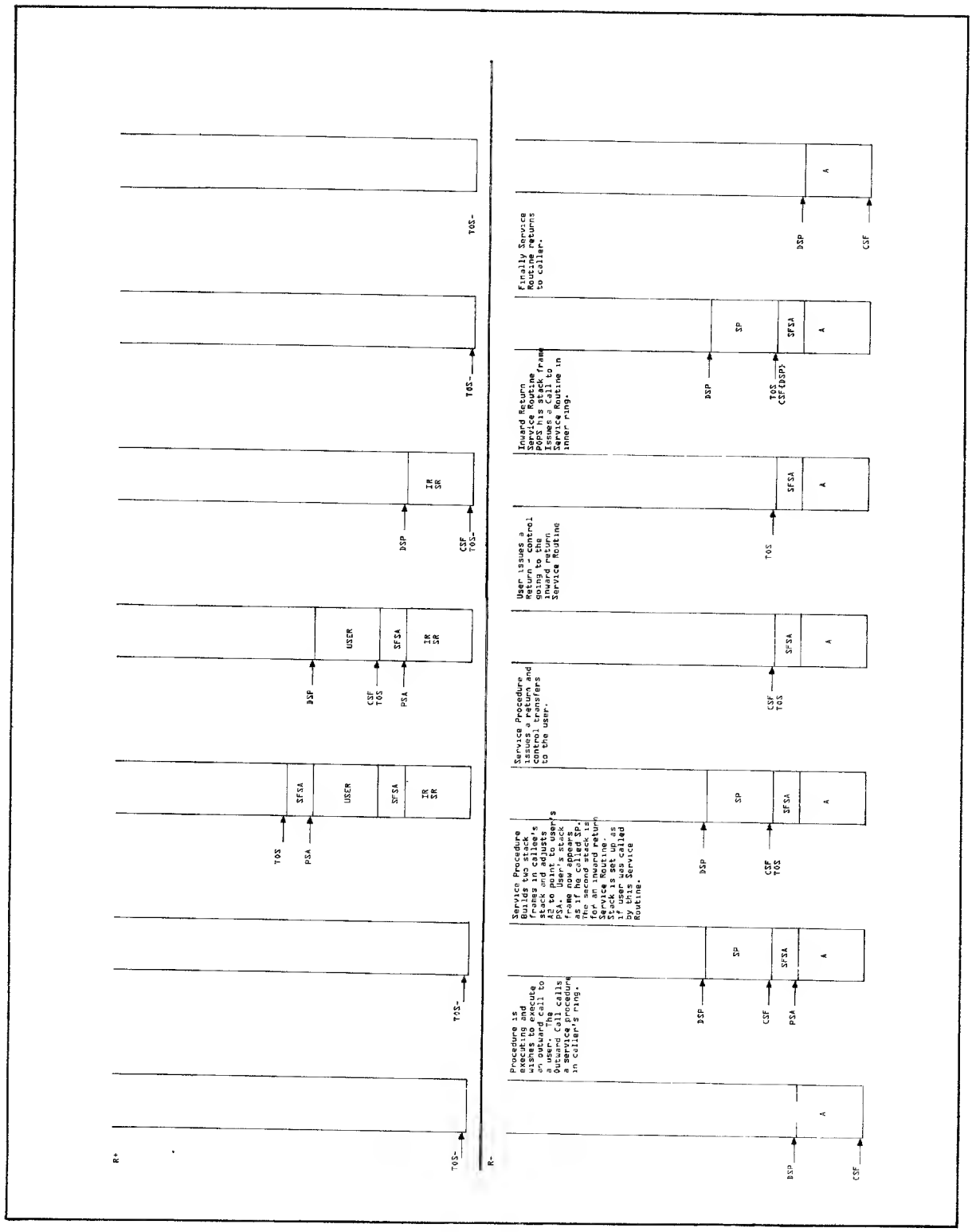


Figure 7-15. OS Call

OBJECT MODULE BINDING

Whenever a procedure is compiled or assembled, directives are compiled with it to enable the loader to create a binding section. In fact, all references to working storage, external procedures, and so forth are compiled as offsets into the binding section. When a program is executed, there are multiple binding sections (one per procedure). There is nothing wrong with this, except that procedures called from several other procedures have an entry in several binding sections. This is a waste of space. Also, many calls to external procedures translate into calls within the same segment (intrasegment calls). These are actually external procedure calls at compile time. They become internal procedures at execute time. The difference is that an external procedure call must be made with a call indirect via the binding section, whereas an internal procedure call can be made with the more efficient call-relative instruction. The object library generator minimizes these space and time inefficiencies.

The object library generator performs two major functions.

- It eliminates redundancy by taking all procedures in a module to be bound and placing them in a single code section. It also combines all binding sections into a single binding section and eliminates redundant entries to external procedures.

- Since many calls to external procedures translate to calls to internal procedures during the coalescing of the binding sections, the call-indirect instructions are converted to call-relative instructions for these procedures, and their entries are eliminated completely from the binding section (figure 7-16).

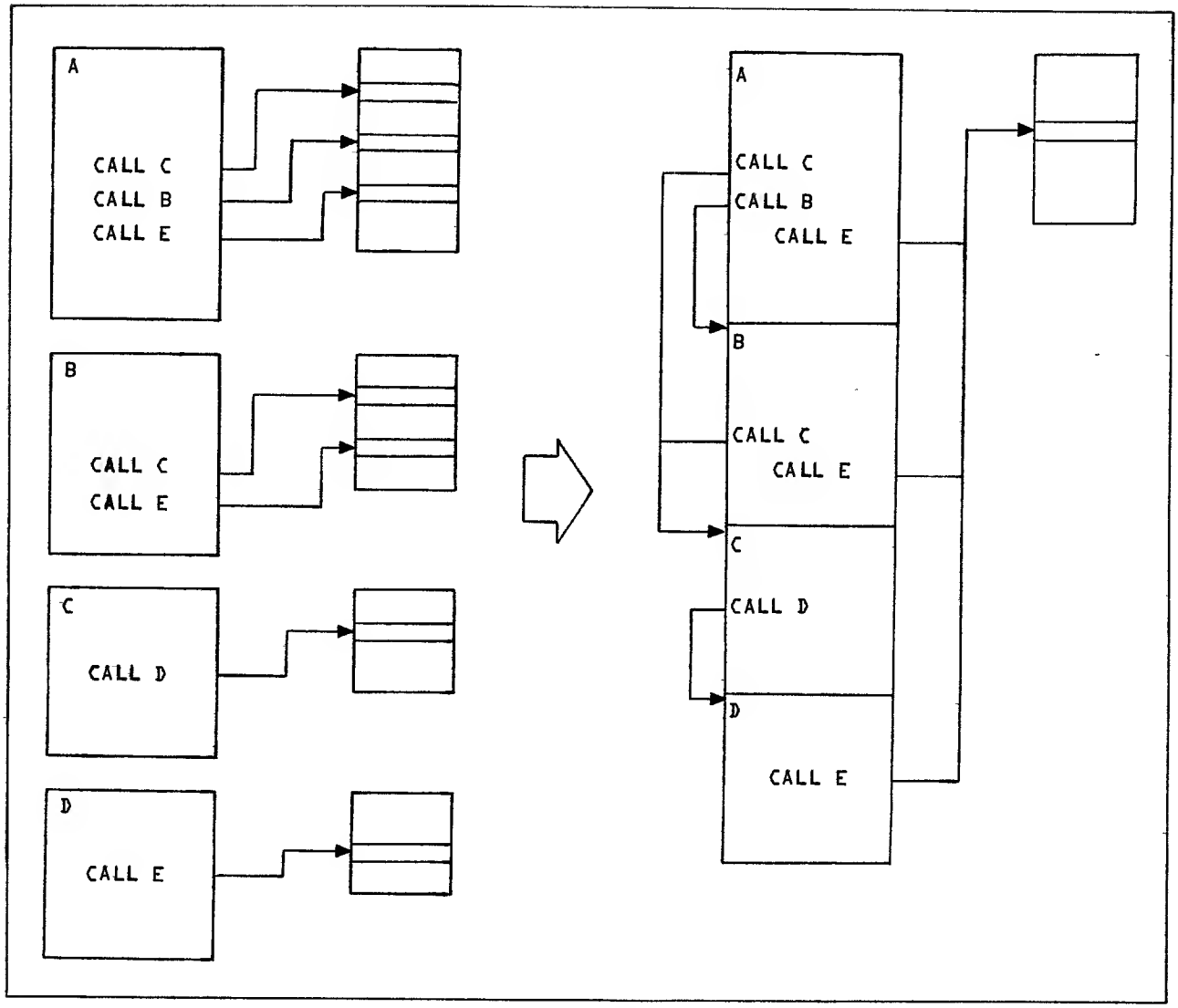


Figure 7-16. Binding Process

This second major function of the object library generator imposes a restriction on the format of the call instructions that have been designed with this purpose in mind. Since they have similar formats, all that need be done by the object library generator is to change the operation code from B5 to B0, set the desired value in the Q field, and force the J field to 3, which is the conventional register for the binding section (figure 7-17).

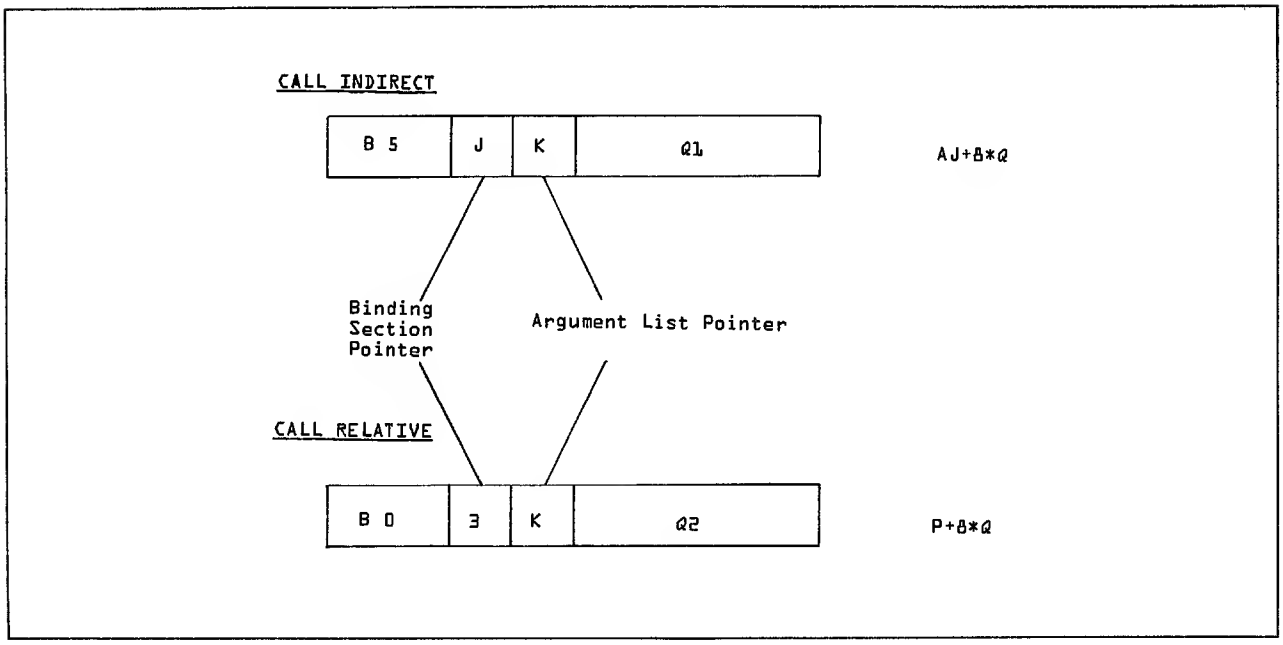


Figure 7-17. Conversion from Call-Indirect to Call-Relative

VIRTUAL MACHINES

Virtual State provides a capability to support several virtual machines. The two most important of these are the native machine (Virtual State) and CYBER 170 State. The call mechanism permits a procedure being executed on one virtual machine to call another procedure that will execute with a different virtual machine. The exact mechanism that accomplishes this machine switch is not described here but is covered in a separate section on virtual machines.

RING NUMBER 0

In the section on virtual memory, it was explained that there are 15 rings of protection on Virtual State. These are numbered 1 through 15. Ring number 0 has been reserved for a special purpose, dynamic linking. Traditionally, a program has been written as a series of subroutines or procedures. These subroutines are compiled separately, then linked with a loader prior to their execution. Depending on the system, all subroutines referenced must be present before the program can be placed in execution. Frequently, this restriction is imposed even though all subroutines are not used. This is one way of solving the problem of linking, loading, and placing a program into execution, but it is by no means the only one. Certainly, this alternative is open to Virtual State and may be a common method chosen by the user. However, Virtual State provides another option. This option is to link and load a procedure the first time it is called and not before. If a procedure is referenced but never called, it need never go through the linking and loading mechanism, and never requires that memory be allocated to it. This process is known as dynamic linking, and a ring number of 0 is reserved to denote an unlinked pointer.

Ring numbers of 0 can occur in one of two ways, either in an attempt to load a pointer into an A register, or in an attempt to call on an unlinked procedure. The Virtual State hardware automatically detects this condition and causes an exchange interrupt to be taken if the machine is in job mode. The exchange interrupt handler can then schedule the appropriate operating system procedure to form the necessary link and to load the required procedure.

When ring number 0 is detected on a load instruction, the following sequence occurs.

1. The load instruction completes, loading the invalid pointer into the appropriate A register with a ring number determined by the normal ring number contention mechanisms (figure 7-18).

2. An exchange interrupt occurs. The P register stored in the exchange package (at JPS) points to the instruction following the load instruction.

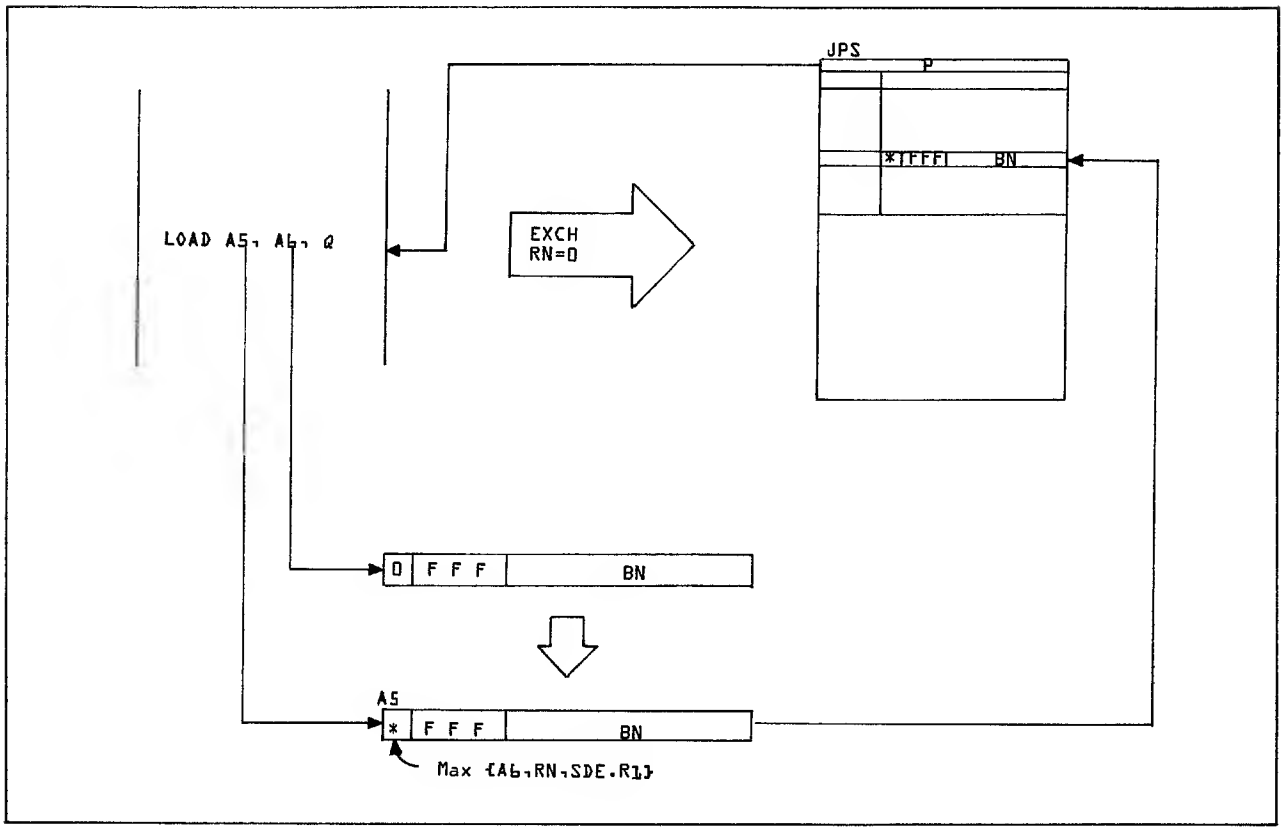


Figure 7-18. Ring Number 0 on Load A

When ring number 0 is detected on a call instruction, the following sequence occurs.

1. The execution of the call instruction is inhibited.

2. An exchange interrupt is taken. The P register stored in the exchange package (at JPS) points to the call instruction in question, and the untranslatable pointer (UTP) register stored in the same exchange package, contains the CBP (with ring number 0) from the binding section that caused the interrupt (figure 7-19).

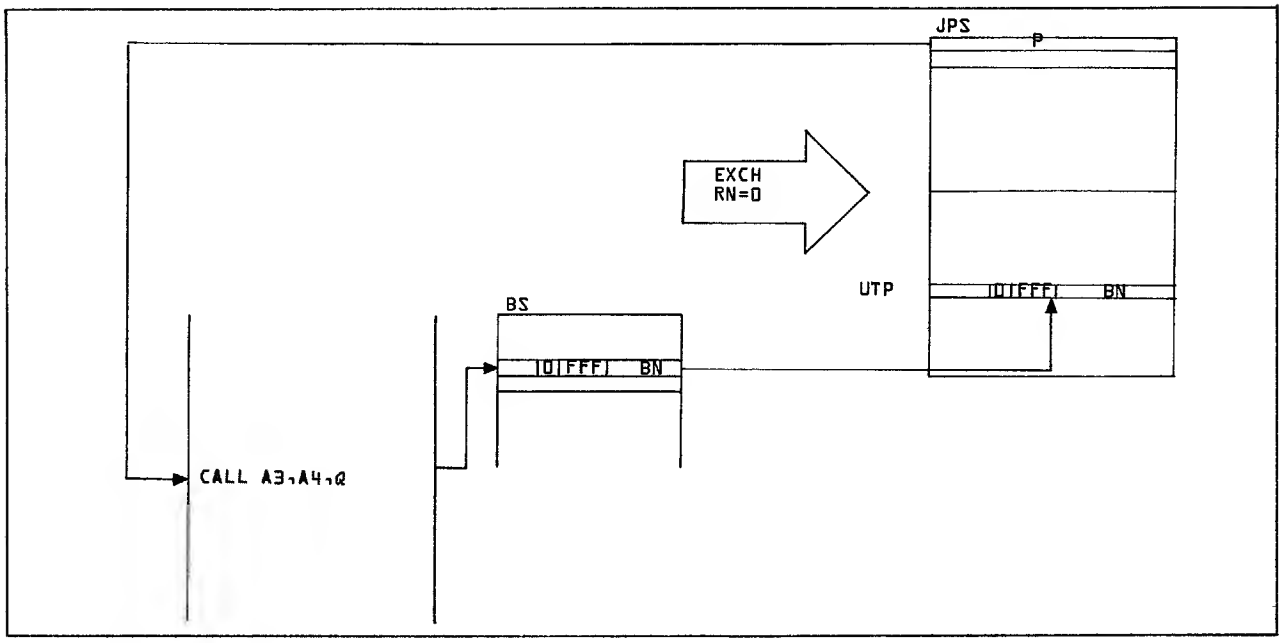


Figure 7-19. Ring Number 0 on Call

The untranslatable pointer is the key to handling this exception combined with an invalid segment exception. On sensing an invalid segment condition, the interrupt handler should check on the UTP. If this has ring number 0 plus a segment number of all 1's, a ring number 0 condition has been detected by a call instruction (as opposed to an invalid segment). The UTP, by software convention, contains a dummy segment number of all 1's (to flag an unlinked pointer) and the byte offset contains pointer to loader tables that contain information necessary to form the required link. The appropriate entry is made in the binding section containing the unlinked code base pointer and an executed exchange jump, which causes the call to be reissued.

If the UTP contained no indication of the fault (this must be established by software convention), the individual A registers (at JPS) must be scanned for the fake segment number. Ring number 0 no longer exists in the register or registers in question, since it is eliminated by the ring number voting mechanism. The register or registers in question are loaded with the correct segment number and byte offset, and an exchange jump is issued to continue processing. Remember to scan all A registers, since several of them could have ring numbers of 0 if a load multiple instruction is used.

Dynamic linking is an option provided by the Virtual State hardware. It provides an alternative to conventional loading techniques. However, there is no need to support this particular technique by software; it is an operating system design decision.

OVERALL PROCESS FLOWCHARTS

Figures 7-20, 7-21, and 7-22 describe the overall process for call, return, and pop. Included in the flowchart for call are those steps unique to a trap interrupt. Remember, a trap interrupt is nothing more than an unsolicited call in which all the A registers and X registers are saved. There are some additional steps. In particular, the condition causing the trap is erased from either the user condition register or the monitor condition register, and those registers are captured in the stack frame save area.

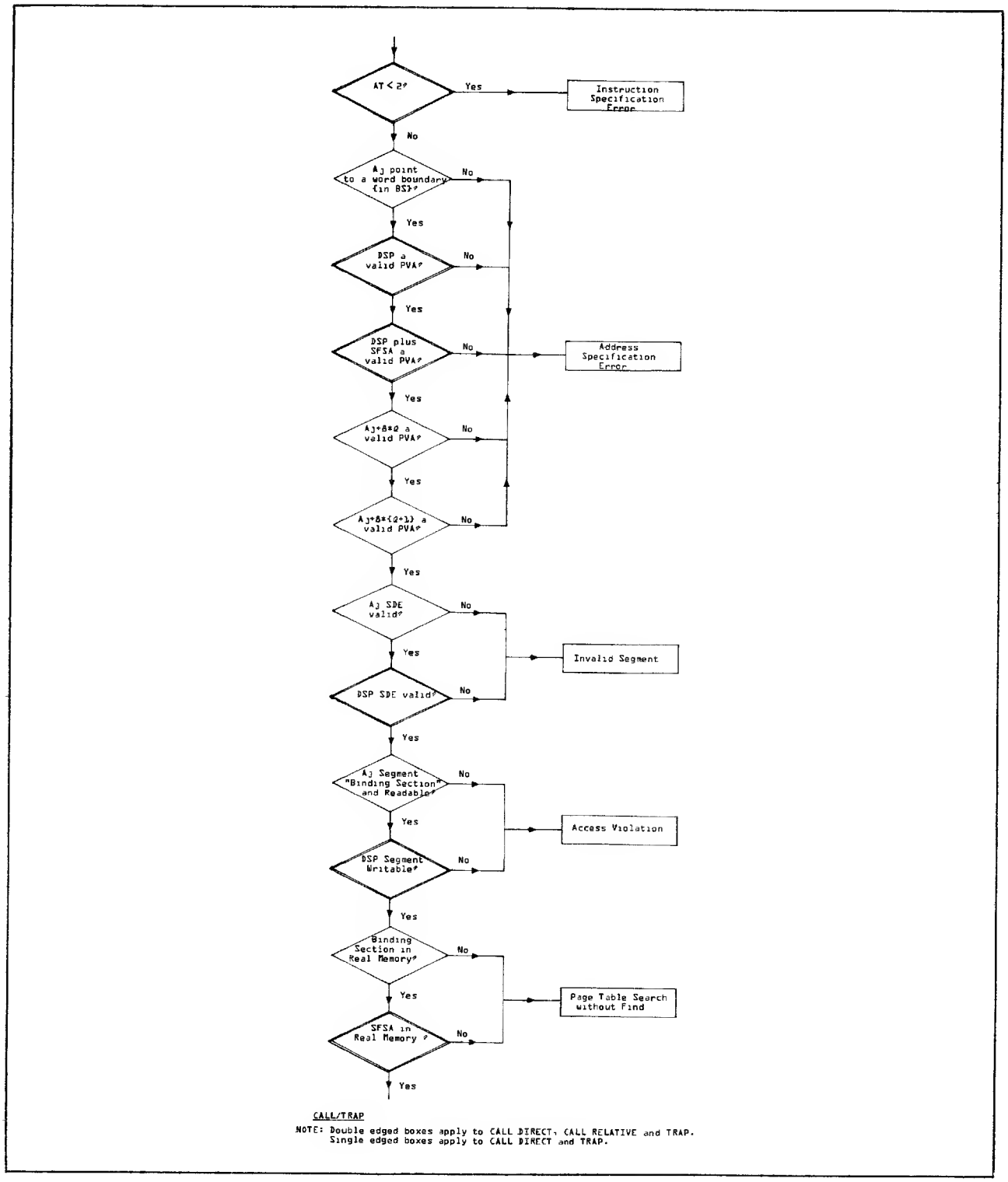


Figure 7-20. Call/Trap (Sheet 1 of 3)

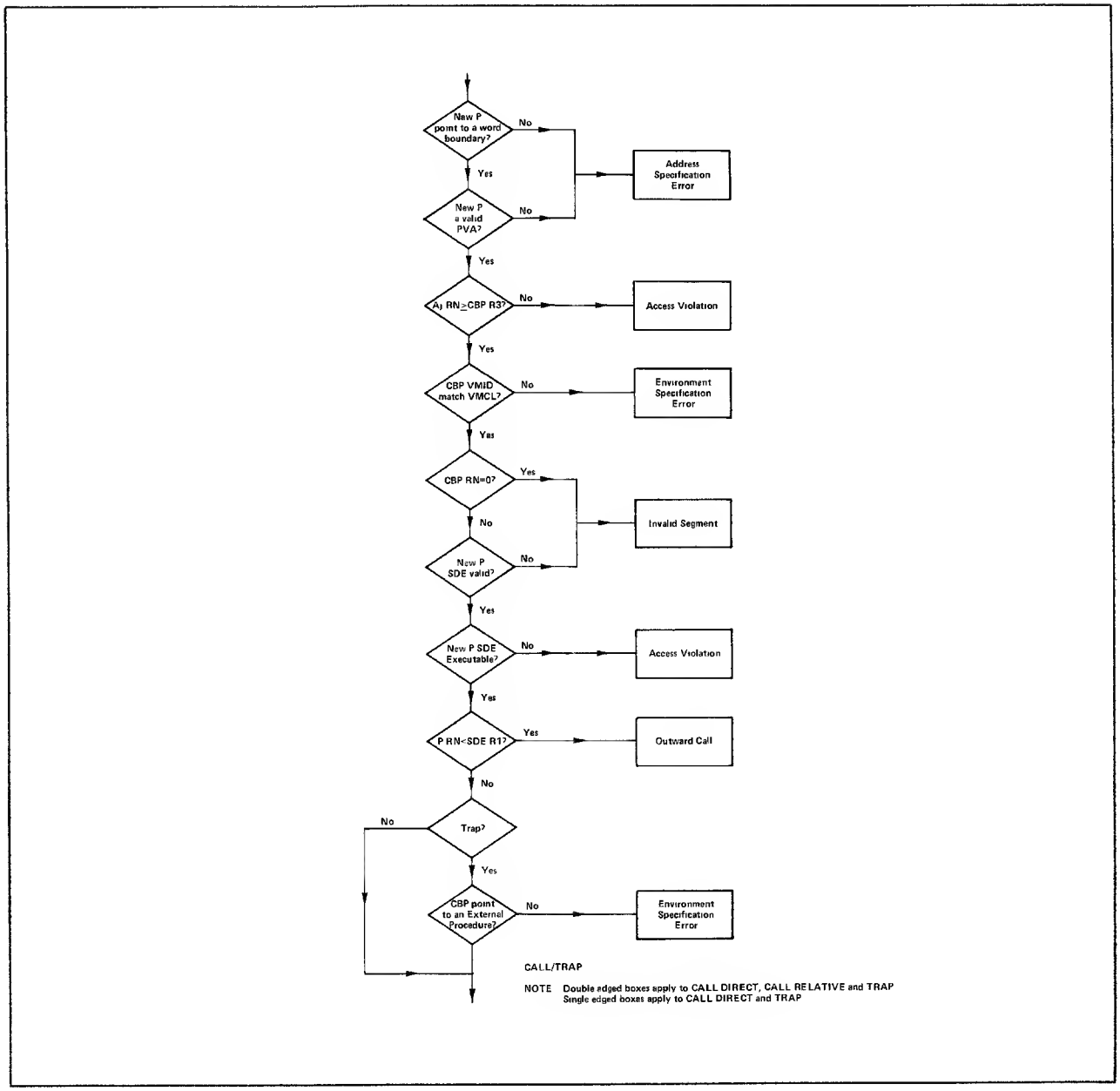


Figure 7-20. Call/Trap (Sheet 2 of 3)

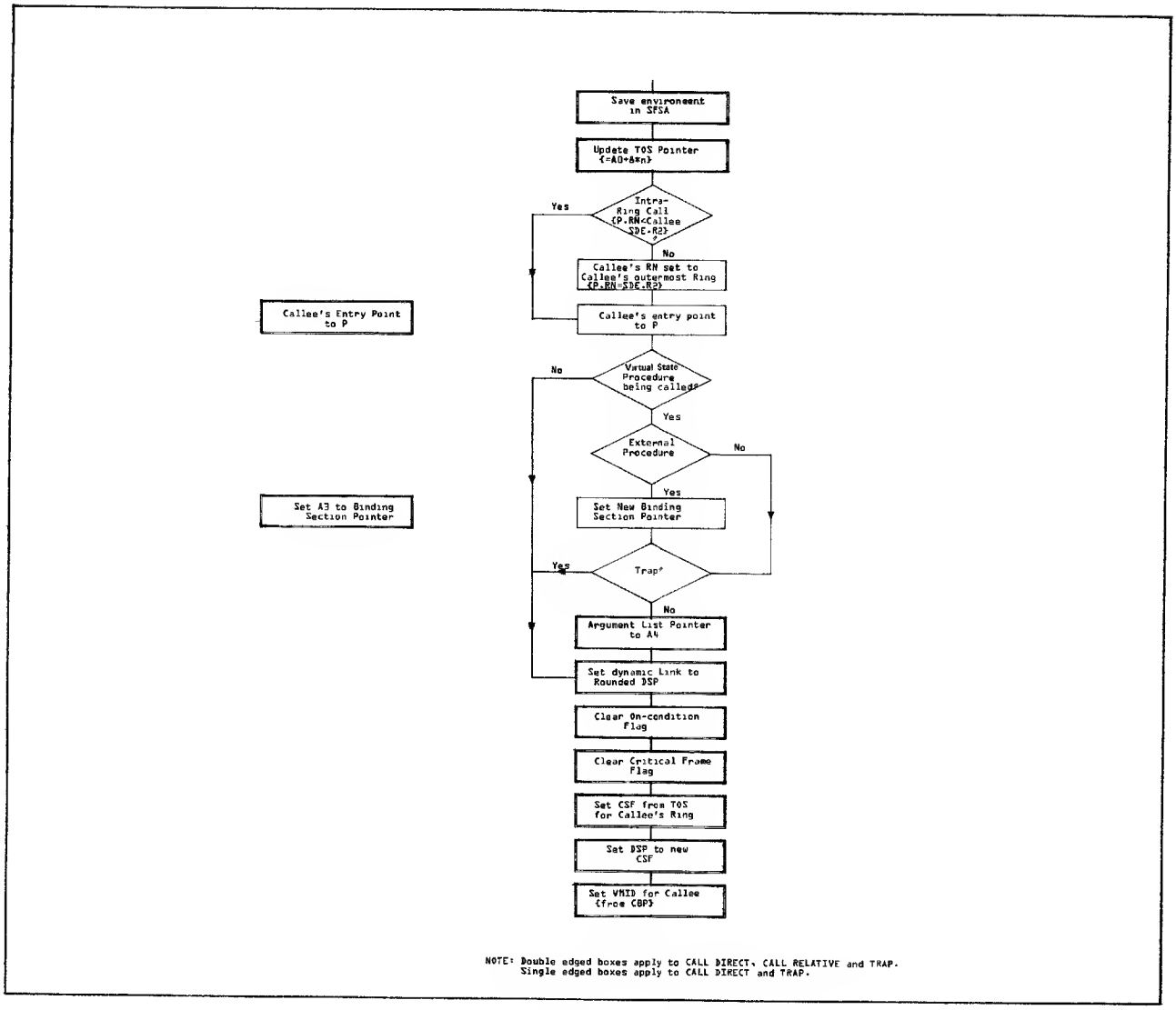


Figure 7-20. Call/Trap (Sheet 3 of 3)

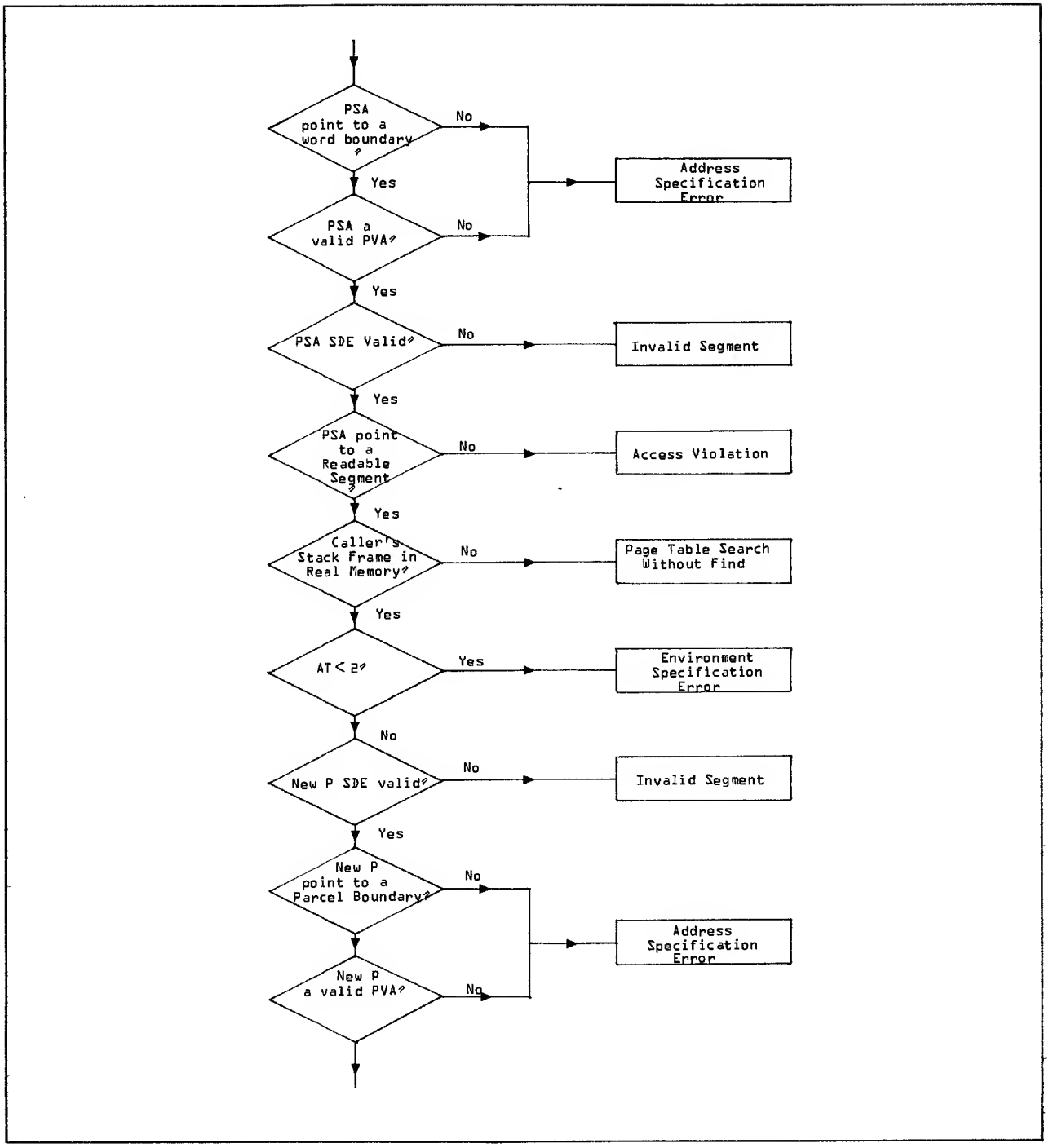


Figure 7-21. Return (Sheet 1 of 2)

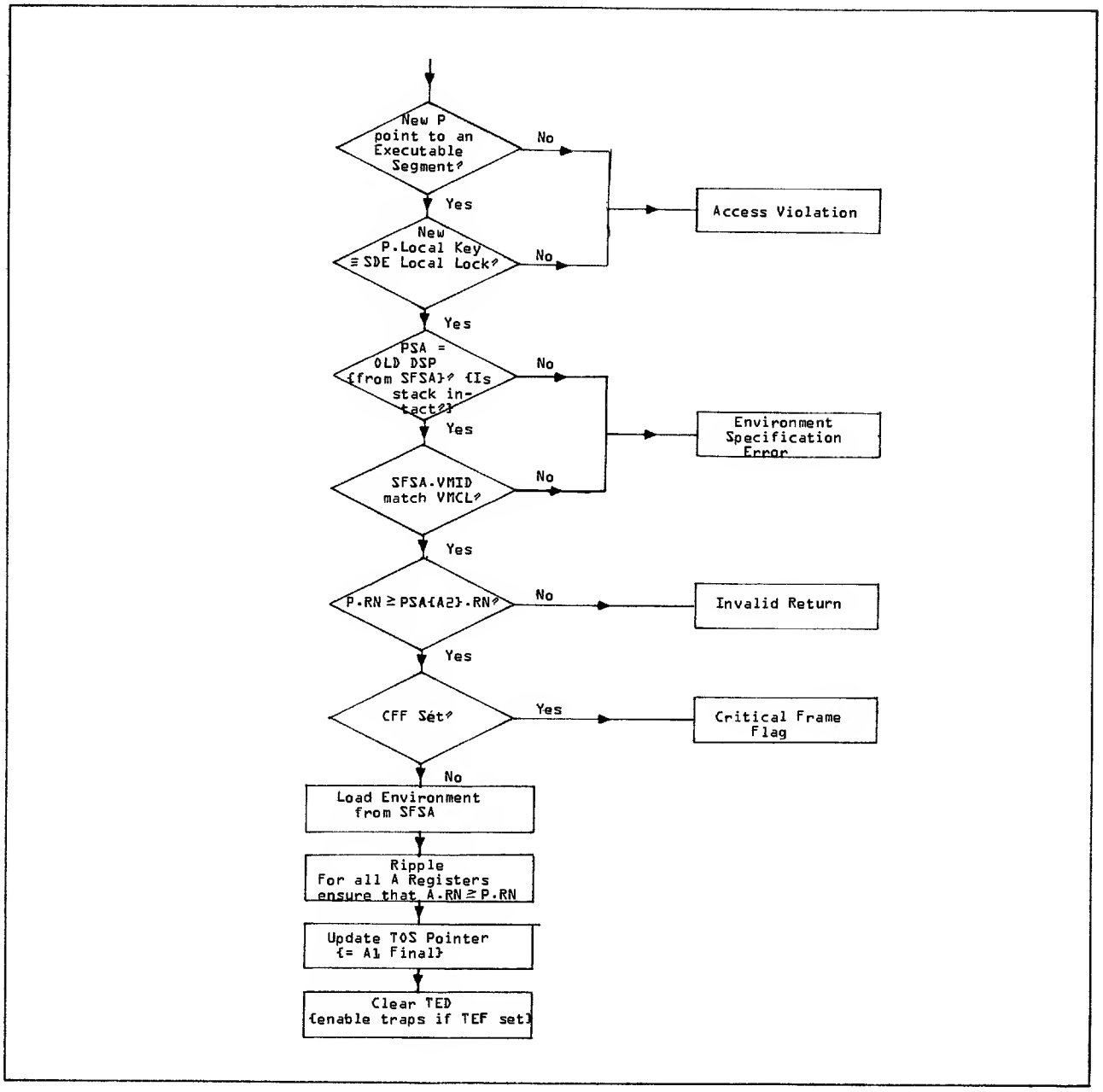


Figure 7-21. Return (Sheet 2 of 2)

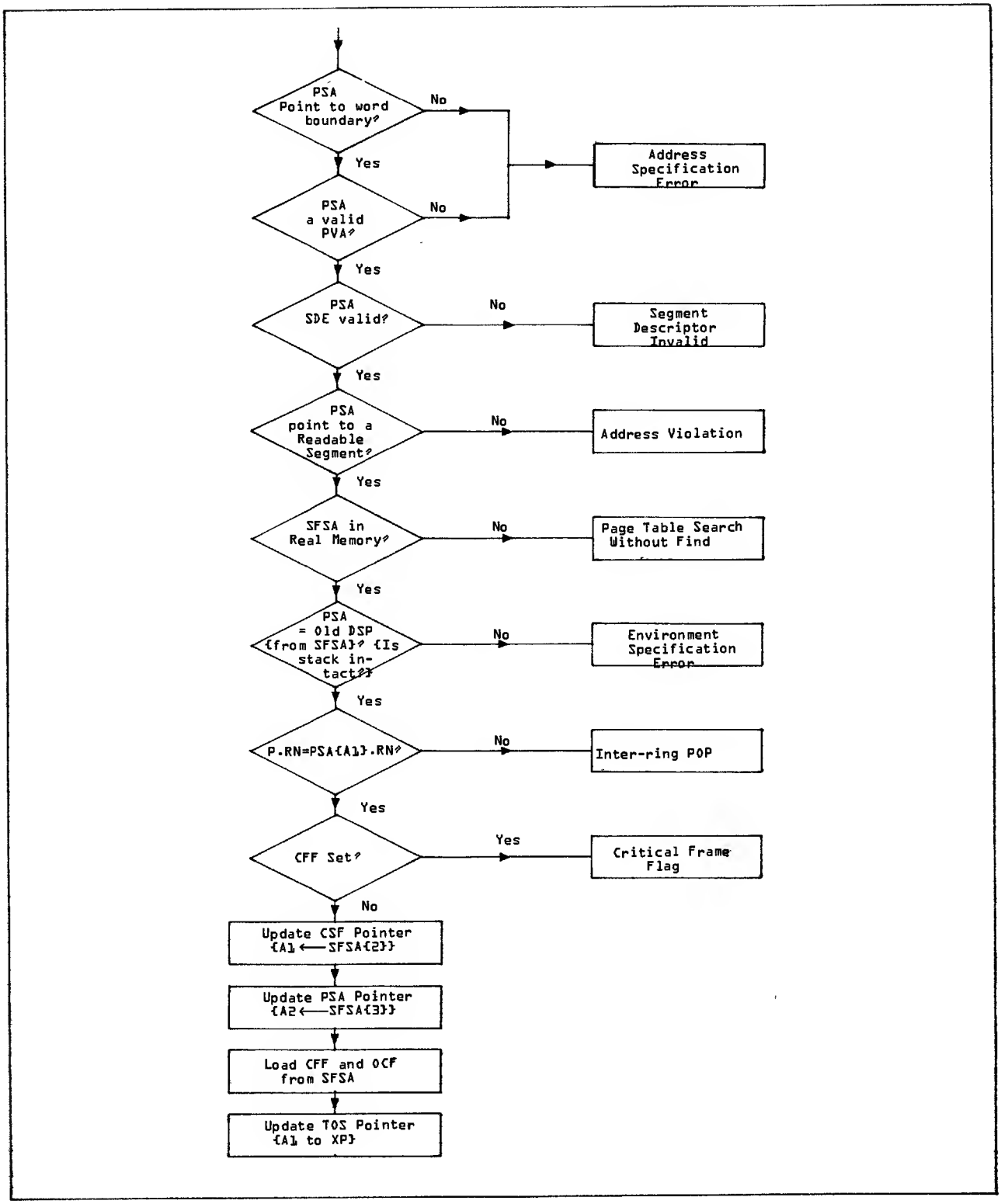


Figure 7-22. Pop

The previous sections described the basic protection mechanisms provided by the Virtual State hardware, including the primary protection afforded by address space, as well as protection mechanisms that are within the address space. It is now necessary to describe the techniques for crossing protection boundaries. Two techniques are available, one for switching between address spaces and one for crossing protection boundaries within an address space.

CHANGING ADDRESS SPACES

The exchange jump is used to transfer control from one address space to another. When an exchange jump occurs, the machine state changes between job mode and monitor mode. This state is controlled by a flip-flop that cannot be cleared or set by software other than by an exchange jump when the flip-flop is complemented. Virtual State processors are always deadstarted into monitor mode via a half exchange. The operating system monitor is the most privileged module of the operating system. It resides in its own address space and has additional special privileges, because it operates in a unique machine state. It is the most trustworthy piece of code in the system. The operating system monitor establishes users' operating environments by defining their exchange packages and, consequently, establishes in part their level of security. This concept of trustworthiness is very important to Virtual State systems. In general, the lower the ring of execution, the more trustworthy is a code module. Virtual State hardware provides the tools necessary to construct a system with any desired level of security. Nevertheless, those hardware facilities are only as good as the software that uses them. For a system to be truly secure, software conventions must be enforced. These conventions form part of the overall architectural design of the system. In concert with this theme, the hardware does very little checking on the operating system monitor. In particular, no ring number checks are performed during an exchange jump. If the monitor elects to increase a user's authority by assigning an A register ring number lower than his or her ring of execution, the user runs with that greater privilege. Because of this and other reasons, the operating system monitor should be an extremely small, thoroughly debugged piece of code.

PROTECTION BOUNDARIES WITHIN AN ADDRESS SPACE

Call/return is the primary mechanism for crossing protection boundaries within an address space. It is the only mechanism for crossing ring boundaries. Two conditions must be satisifed before crossing a protection boundary. First, the caller must be permitted to make the call; second, the callee must not act on behalf of caller with more authority than caller. (Call/return is described in detail in section 7.)

If a user tries to make a call to a more privileged ring than he or she has the right to use, a potential breach in security occurs. The hardware prevents this occurrence by detecting attempts either to call outward to a ring of less privilege, or to return inward to a ring of more privilege. Such an attempted breach in security causes an exchange interrupt into the monitor address space. The details of security and protection are discussed in section 3.

Most of the information pertaining to security is managed by hardware and is contained in hardware tables, although the tables are constructed by software. The main such table is the segment descriptor table (SDT). Whenever a call is made to another segment across a protection boundary, the transfer must take place in a controlled manner. To accomplish this, calls across protection boundaries do not take place directly, but instead use an indirect or process virtual address (PVA) held in a pointer in a binding section. By software convention, binding sections are not writable in user rings, and are constructed by the loader based on directives issued by compilers and assemblers. The hardware ensures that all calls across protection boundaries take place via a binding section entry. An access violation interrupt causes an exchange to monitor mode if an attempt is made to bypass this mechanism.

Many other security checks are performed by the hardware during a call. Some are fairly straightforward. For example:

- The SFSA must be in a segment that has write permission.
- The callee's entry point (obtained from the binding section) must be in a segment that has execute permission.

The hardware also ensures that the caller is within the callee's call bracket, as described in the discussion of rings of protection. The pointer to the callee's entry point in the binding section is named a code base pointer (CBP) and has the format shown in figure 8-1.

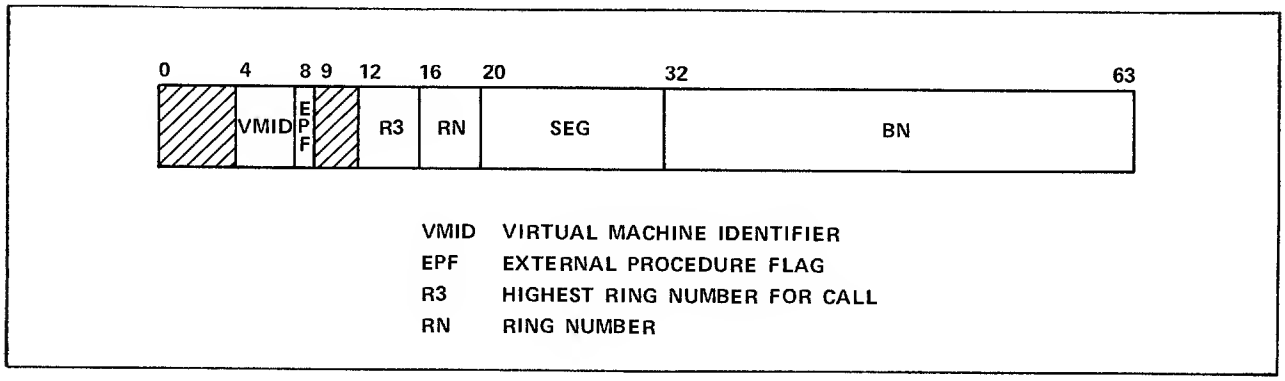


Figure 8-1. Code Base Pointer (CBP)

A call is permitted, providing:

$$PVA.RN \leq CBP.R3$$

The first check performed by the hardware during a call ensures that the caller's ring number (held in the P register) is within the callee's call bracket. That is:

$$P.RN \leq CBP.R3$$

In practice this check is made implicitly. An explicit check is made against the Aj ring number as described below. Of itself this check is insufficient, since a caller could ask a more privileged procedure to call on his behalf a third procedure to which he does not normally have access.

In figure 8-2, procedure A resides in ring 13 and procedure B resides in ring 11.

INTERSEGMENT BRANCH

It was mentioned earlier that the call/return mechanism is the primary mechanism employed for crossing protection boundaries. It is the only mechanism available for crossing rings. However, another instruction, intersegment branch, can be used to transfer control from one segment to another. Since such a transfer of control involves crossing a key/lock protection boundary, the hardware must ensure that the correct key/lock transformations occur. The execution of this instruction is illustrated in figure 8-3.

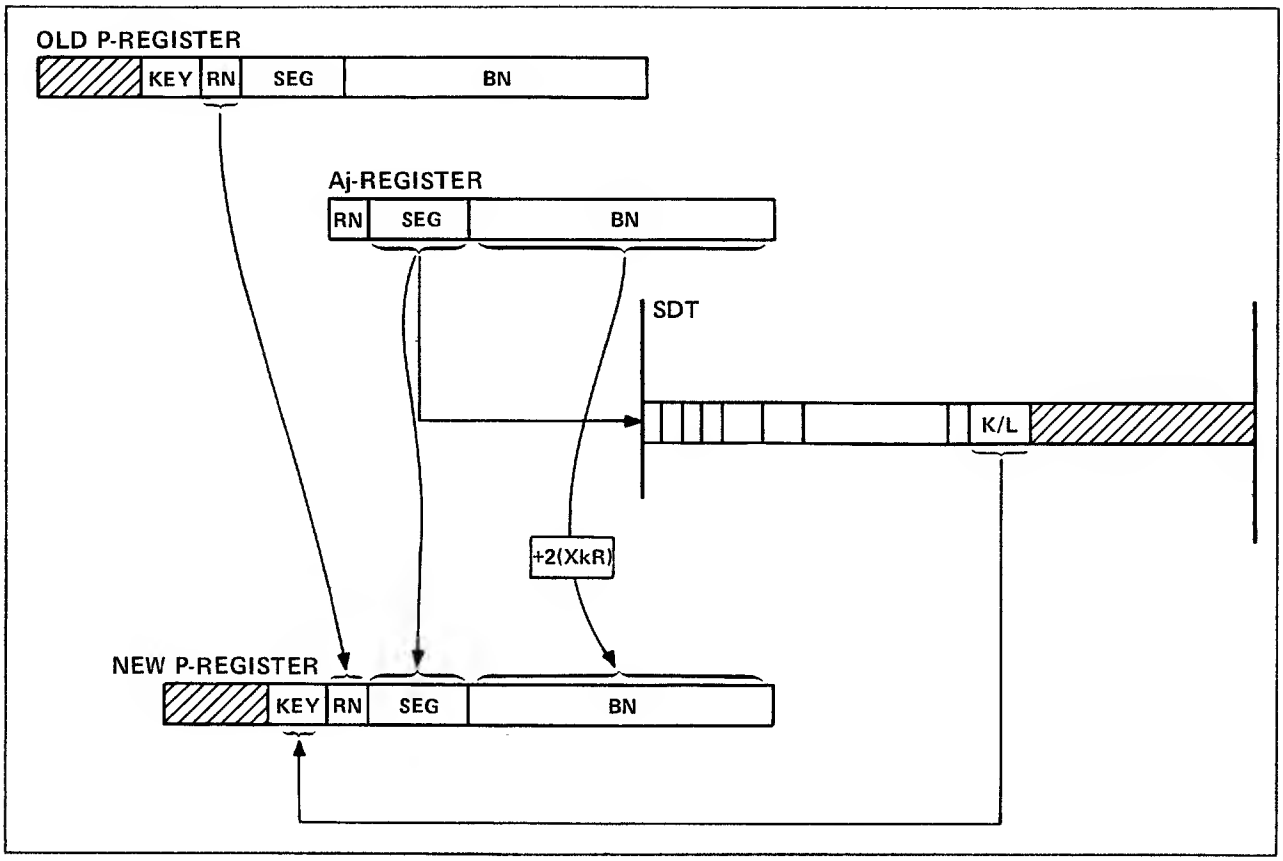


Figure 8-3. Intersegment Branch

Notice that the new P register ring number is forced to the value in the old P register. Ring boundaries cannot be crossed by this instruction. In addition, the new P register key is taken from the associated SDE lock. An executing procedure always runs with its own key.

WHEN HARDWARE CHECKS OCCUR

The hardware makes the following checks for access violations on each occurrence of the following actions.

Read access to a segment:

- The segment must have read access.
- The segment must be readable from the ring of the procedure making the access (this is via the ring number of the A register used to make this access).
- The current key exactly equals the lock of the segment, in the absence of a master key or no lock.

Write access to a segment:

- The segment must have write access.
- The segment must be writable from the ring of the procedure making the access (this is via the ring number of the A register used to make the access).
- The current key exactly equals the lock of the segment, in the absence of a master key or no lock.

Call to an external procedure:

- The CBP must be in a binding section.
- The current SFSA must be in a segment that has write access.
- The procedure being called must be in a segment that has execute access.
- The caller must be within the callee's call bracket.
- The call must not be an outward call.

Return from an external procedure:

- The previous SFSA must be in a segment that has read access.
- The procedure that control is returned to must be in a segment that has execute access.
- The final key (obtained from the P register in the SFSA) equals the associated segment's (caller's) lock.
- The return must be an outward return.

In addition, for each A register ring number that is less than the final P register ring number, the associated A registers ring number is set equal to the P register ring number.

First instruction issued from a new segment:

- The segment must have execute access. This check is not repeated for further instructions issued from the same segment. Normally, the check occurs during the execution of the instruction that transferred control to the new segment, either during the call or intersegment branch.

These are not the only checks performed by the hardware during the execution of these instructions. These are just the checks made to ensure that an access violation is not being attempted. Many other checks are made to ensure that the hardware functions correctly. For example, all branches must be to parcel boundaries, and all calls must be to word boundaries.

SOFTWARE CONVENTIONS

The hardware provides the mechanism necessary to construct a secure system. However, it is the software use of the hardware that determines the ultimate level of security. For the system to be completely secure, the software must adhere to several conventions. Some of these have been discussed in the previous sections. They are now summarized in this section.

RINGS OF PROTECTION

Since the ring protection mechanism is hierarchical, the higher the privilege assigned to a procedure, that is, the lower the ring number, the more trustworthy that procedure must be. The more privileged a procedure, the more thoroughly it must be checked. The operating system monitor, which is the most privileged procedure in the system, should be kept as small as possible and thoroughly checked. Also, the more privileged procedure must always ensure that its own integrity is not jeopardized. In particular, care must be exercised when a procedure acts on behalf of a less privileged procedure. In this case, whenever data is referenced via the caller's arguments, the callee must reference this data through directly loaded A registers. That is, the callee must ensure that the hardware A register ring voting is exercised whenever the caller's pointers are used. This is in place of a pointer being loaded in an X register and then being switched into an A register (using a copy X to A instruction), when the callee's ring number could result in the caller's pointer. Since most software is developed in a high-level language, the compilers must adhere to this convention.

CONTROLLING PROCEDURES

As has already been described, much of the security of the system is ensured by the hardware. The hardware utilizes various hardware tables, in particular, the segment descriptor table (SDT). These tables are constructed by software procedures. These procedures are trustworthy, and they will execute in low-numbered rings, but not necessarily in ring 1. They should be developed in such a way that they are self-contained, as small as possible, and impossible to tamper with, unless the most stringent security checks have been taken and passed. The security mechanisms that have already been described take care of security problems when the procedures are being executed. However, when they are modified, either statically or dynamically, a combination of installation procedures and operating system services must be brought into play to ensure that the security of the system is maintained.

USER RESPONSIBILITIES

The hardware and software mechanisms that interplay to provide system-wide security and protection have been described. At first glance, it may appear that the utilization of these facilities places a heavy burden on the end user. Fortunately, this is not the case, although a responsibility is placed on the installation management. Much of the security of the system is centered on the operating system file system. Every file carries with it the four ring brackets, for read, write, execute and call, that have already been described. Assignment of these ring brackets is based on the privilege the user has been validated for. Before a user can log in to the system, in either batch or interactive mode, that user must be known to the system. He or she enters a user number and a password as identification. These parameters direct the system to a validation file containing the privileges of the user.

The typical end user should be totally unaware of his ring of execution and whether or not his code and data segments carry nonzero locks. If the user desires to protect some local data, suitable directives to the operating system cause the setting of the appropriate lock values. Again, the actual value of these locks is of no concern to the user. Consequently, the typical end user, who is, for example, running FORTRAN codes, need not be concerned with the security mechanisms of the system. At the same time, these mechanisms are in play to isolate him from other users and from the system.

The section on call/return should be thoroughly understood before proceeding with this section. When the subject of interrupts was introduced, their hierarchical nature was described. This hierarchy involved two types of interrupts, exchange interrupts and trap interrupts. In an exchange interrupt, the state of the machine changes from job mode to monitor mode. All process state registers are saved in one area of memory and loaded from another area. Included in the process state registers is the P register, and execution continues after the exchange at the address pointed to by the P register.

In a trap interrupt, although the purpose is similar (that is, to stop the normal sequence of operation and transfer control to another instruction sequence in such a way that the original sequence can be restarted at the point that it was interrupted), the mechanism is quite different. In fact, a trap interrupt is an implicit call. Not all the process state registers are saved and very few are loaded with different values. A maximum SFSA is created, and all A registers and X registers are saved in it, along with other key process state registers. The P register is saved, and processing continues at the address given by a code base pointer (CBP) in the binding section of the interrupted process. The address of this CBP is given by the trap pointer, and the CBP must point to an external procedure for the trap interrupt to complete.

Trap interrupts, therefore, transfer control to an address within the address space of the executing process. This is important, because the trap handler normally has to make reference to flags and data held in the user's stack. In fact, the outward-call/inward-return mechanism described in the last section is conducted primarily in the user address space, even though it is initiated by an exchange interrupt. The free-flag is used to cause an interrupt to take place in user's address space.

Of major importance to the trap interrupt operation is the management of the condition registers and trap control flags, both the trap enable flip-flop (TEF) and trap enable delay (TED). When the trap interrupt is taken, the user and monitor condition registers are stored in the SFSA and the bit (or bits) that causes the interrupt is cleared from the appropriate condition register. These registers are reset on the trap and can start collecting new fault conditions in an unambiguous manner. Also, when the trap interrupt is taken, traps are disabled and the TEF is cleared.

To reenable interrupts, two mechanisms are available. Setting the TEF via a copy instruction accomplishes this. However, this is not the normal technique used. The trap interrupt is an implicit call, and the continuation of normal processing is accomplished by a return instruction. Part of the return mechanism reenables interrupts. The sequence of events is to set the TEF and the TED (by a single copy instruction), then to issue the return. When the TED is set, traps are disabled regardless of the setting of the TEF. The return instruction clears the TED which, if the TEF is set, reenables interrupts. Since the TED is cleared only upon completion of the return instruction, problems associated with enabling traps in one instruction step, then returning in a second step, are avoided.

INTERRUPT CONDITIONS

Now that the basic interrupt mechanism has been described, we can proceed to the individual interrupt conditions. Virtual State interrupts are precise. That is, the interrupt handler can always refer back exactly to the instruction that caused the interrupt, or that was being executed when the interrupt occurred. However, depending on the nature of the interrupt, the method for tracing back to the instruction in question varies.

A basic architectural philosophy of Virtual State is that an instruction is not interrupted during its execution. Conditions that prevent an instruction from executing are checked before the instruction is committed. The concept of a point of no return was introduced in an earlier section on interrupts and this is an important concept. Any exception conditions detected before the point of no return prevent the instruction from executing, an interrupt is taken, and the P Register, at the time of the interrupt, points to the instruction that could not be executed.

MONITOR CONDITION REGISTER (MCR)

Figure 9-1 lists the conditions recorded in the Monitor Condition Register. Following are some notes on these conditions.

P REG	BIT NUMBER AND DEFINITION			ASSOCIATED MONITOR MASK REGISTER BIT SET				MASK BIT CLEAR
				TRAP ENABLED		TRAP DISABLED		TRAP ENABLED OR DISABLED
				JOB MODE	MONITOR MODE	JOB MODE	MONITOR MODE	JOB OR MONITOR MODE
–	48	Detected Uncorrectable Error	Mon	EXCH	TRAP	EXCH	HALT	HALT
–	49	Unassigned		EXCH	TRAP	EXCH	HALT	HALT
P+	50	Short Warning	Sys	EXCH	TRAP	EXCH	STACK	STACK
P	51	Instruction Specification Error	Mon	EXCH	TRAP	EXCH	HALT	HALT
P	52	Address Specification Error	Mon	EXCH	TRAP	EXCH	HALT	HALT
P+	53	170 Exchange Request	Sys	EXCH	TRAP	EXCH	STACK	STACK
P	54	Access Violation	Mon	EXCH	TRAP	EXCH	HALT	HALT
P	55	Environment Specification Error	Mon	EXCH	TRAP	EXCH	HALT	HALT
P+	56	External Interrupt	Sys	EXCH	TRAP	EXCH	STACK	STACK
P	57	Page Table Search Without Find	Mon	EXCH	TRAP	EXCH	HALT	HALT
P+	58	System Call	Status - This bit is a flag only and does not cause any hardware action.					
P+	59	System Interval Timer	Sys	EXCH	TRAP	EXCH	STACK	STACK
P/P+*	60	Invalid Segment/Ring Number Zero	Mon	EXCH	TRAP	EXCH	HALT	HALT
P	61	Outward Call/Inward Return	Mon	EXCH	TRAP	EXCH	HALT	HALT
P+	62	Soft Error Log	Sys	EXCH	TRAP	EXCH	STACK	STACK
–	63	Trap Exception	Status - This bit is a flag only and does not cause any hardware action.					

* P, unless P+ for RNO on loads

Figure 9-1. Monitor Condition Register

Detected Uncorrectable Error (DUE)

This interrupt indicates that an uncorrectable error has been detected in either the processor or the memory on a reference generated by the processor.

Major data paths, registers, or control memories all carry either parity or SECDED. Any error detected before the point of no return of an instruction causes the instruction to be retried. A retry counter (one counter that applies to all errors) may be set. If the instruction retry is unsuccessful, a detected uncorrectable error (DUE) is recorded. The P register, saved by the interrupt, points to the instruction that was in execution at the time the interrupt occurred, if it occurred before the point of no return. If the error arose after the point of no return, then it is handled by the complete portion of the instruction execution. In this case, the P register saved by the interrupt points to the instruction following the one that was in execution when the error was detected. This means that there is no way of resuming the instruction stream after the interrupt. However, since the state of the process that was executing is undefined, there is little point in doing this.

To aid in recovering processors, the processor not damaged (PND) flag is set if the fault occurred before the point of no return. When this flag is set, the process environment is intact, even though further processing may be impossible. This fact may be used by the damage assessor to subsequently restart the process.

Memory malfunctions are included in this condition. An understanding of the types of errors that can arise in memory may help in an assessment of the best way to handle them. A simplified picture of memory error detection is shown in figure 9-2. Data transmissions between a processor and memory are checked for correct parity at the processor port, at the memory port, and at the memory array paks. Memory itself, such as chips and bank logic, has SECDED.

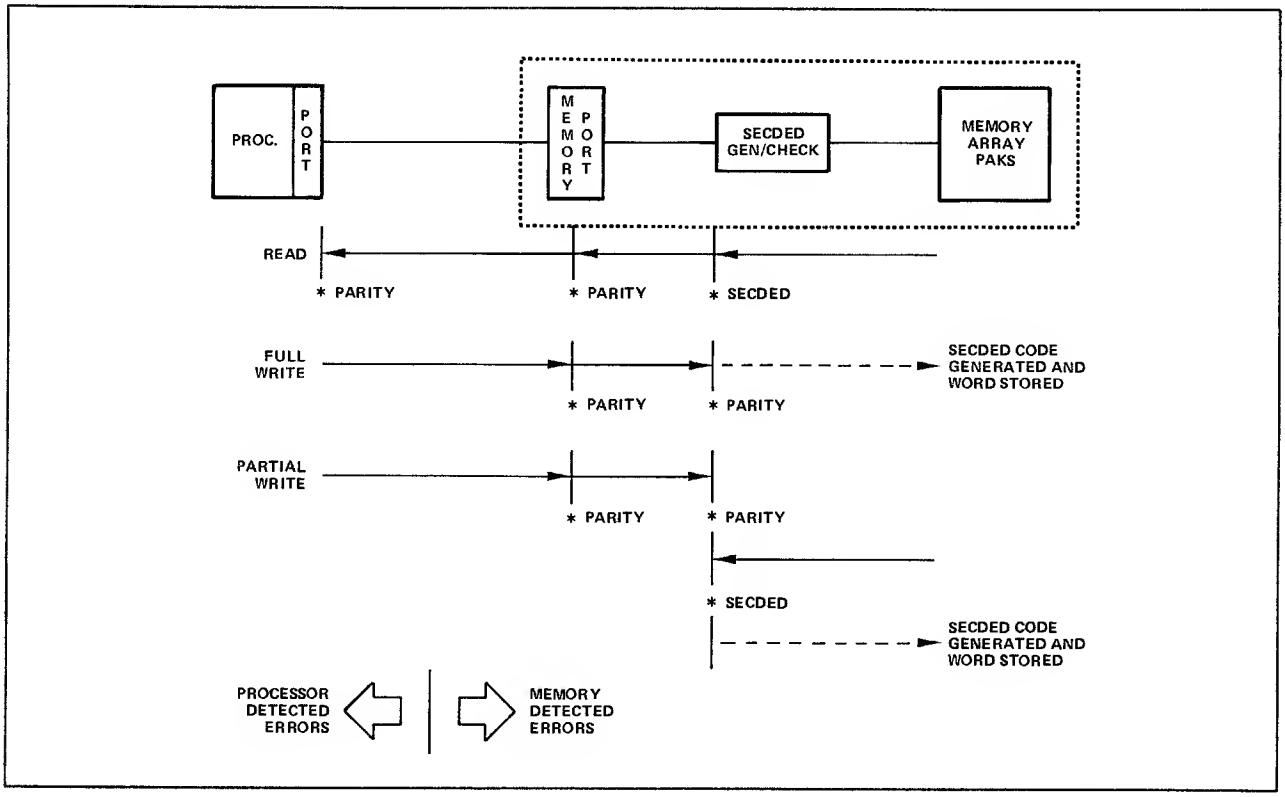


Figure 9-2. Memory Error Detection

A read request is essentially a synchronous process. The requesting processor must wait for the data transmission to complete. The transmission does a SECDED check and then is parity-checked at the memory port before being routed to the processor. Errors detected up to this point result in a memory-detected malfunction. The transmission is then parity-checked at the processor port, and an error here results in a processor-detected malfunction. The now incorrect data continues to its destination and the process in execution is handled as previously described.

Two forms of write request are of interest. These forms are a partial write, in which only a portion of a 64-bit central memory word is written, and a full-word write, in which a 64-bit word is stored in central memory.

On a partial write, the word being modified must first be fetched from central memory, and then be rewritten. The data transmission is checked for parity at the memory port and again at the memory array paks (actually at the SECDED generator). The word to be modified is then fetched and checked for SECDED. Finally it is updated, has a new SECDED code generated for it, and is saved in central memory. Any detected error is recorded as a memory detected malfunction.

On a full-word write the sequence is the same as for the partial write, except that the steps where the word is read from central memory and is updated are omitted. On a full-word write, only parity errors can be detected.

A write request is essentially an asynchronous event. The processor issues the request to memory and continues processing. Any errors detected by memory are reported to the processor at a point in the processing that is not associated with the failed write operation. It is virtually impossible to relate to the instruction that is affected by the error. It is pointless, therefore, to continue execution of the process in question.

It is not a bad strategy when the errors are encountered to assume that a user job was being processed, and to attempt to take the interrupt. If the error was transient or in a part of the machine that can be bypassed, the task can be aborted and processing can proceed, maybe after an appropriate reconfiguration has taken place. If a second occurrence of the failure is encountered during the interrupt, the processor either tries to trap or halts. In the extreme case, the processor halts.

When a DUE is present there may be other bits set in the MCR/UCR as a result of the error, all of which should be disregarded.

Not Assigned

This bit is not set implicitly by any hardware condition, but may be set or cleared explicitly by software on exchange or branch on condition register as any other condition register bit. When set explicitly, this bit causes program interruptions in a manner identical to bit 48 of the MCR.

Short Warning

A short warning interrupt is one of several asynchronous interrupts (external events) that may arise. In all cases like these, the P register saved by the interrupt points to the next instruction in sequence to be executed. In other words, it is always detected at the next encountered point of no return. A short warning interrupt indicates that within a minimum of 2.5 seconds, a system critical component will fail and will automatically shut itself down. The operating system must take the necessary steps within this timeframe to ensure an orderly restart. System critical components include, as a minimum, the MG set (main power supply) and all mainframe elements (processors, memories, and the IOU). In addition, customers have an option to purchase a configuration environment monitor (CEM). This monitor detects and reports impending shutdowns in key peripheral equipment, such as the system disk(s) and controller(s). The short-warning interrupt signals an impending shutdown of key equipment. It may be a power failure, but it could be a high-temperature condition, or some other condition likely to cause damage to the equipment unless prompt action is taken. This interrupt never causes the processor to halt.

The short-warning bit in the MCR remains set as long as the condition holds. Even though an exchange interrupt occurs, and a new copy of the MCR is obtained, the power warning bit remains set. If the operating system monitor is entered with the traps enabled, an immediate trap results.

There is a second indication of a short warning intended for use in CYBER 170 State. A bit is reserved for that purpose in the processor status summary register. The process of recording the condition is basically the same as that for the MCR. As long as the situation holds, the condition remains recorded in the status summary register. In the event that it clears (a transient power loss), the condition goes away. This enables software to monitor for restart conditions.

Instruction Specification Error

The instruction specification error is one of a class of errors where the user has either made an error (such as executing data), or is deliberately trying to tamper with the system. In either event, an exchange interrupt is taken with the P register, saved at JPS, pointing to the instruction that caused the error. The only case where a user may deliberately be trying to destroy the system is when he or she attempts to execute a monitor instruction in job mode. The interrupt enables the operating system to abort the job, and to report to the end-user the precise instruction and address, within the process being executed, that caused the fault.

The special instructions for use only by monitor are described in volume II of the Virtual State hardware reference manual (refer to the preface for more information).

Address Specification Error

Certain instructions require the use of a particular form of an address. If the required form is not used, the address specification error occurs, and the operating system may follow the actions suggested for an instruction specification error. Here also the P register, saved at JPS, points to the instruction with the faulty address. In addition, the faulty address is loaded into the untranslatable pointer register (UTP).

CYBER 170 State Exchange Request

Virtual State is designed so it can execute the instructions not only of Virtual State, but of other machines as well. CYBER 170 State is the most important of these. On CYBER 170 State the IOU can initiate an exchange jump in the CPU. However, when this happens on Virtual State, it can only be executed if the CYBER 170 State virtual machine is being executed. If the Virtual State virtual machine is being executed, then an exchange request interrupt occurs and the Virtual State monitor must exchange to the CYBER 170 State virtual machine in order for the request to be satisfied. This is an asynchronous interrupt and the P register, stored at JPS by the interrupt, is set accordingly.

Access Violation

Access violation occurs when a user attempts to access code or data to which he or she has not been granted access privilege. The Virtual State protection mechanism is described fully in the section dealing with virtual memory. It is a mechanism built into the hardware, and any attempt to circumvent it leads to this interrupt. This is the same as an instruction specification error, in that the P register saved at JPS points to the instruction that attempted to violate the protection mechanism. In addition, the address that caused the access violation is saved in the UTP.

Environment Specification Error

An environment specification error indicates that the environment has been destroyed in some way, typically by a programming error. The destruction is such that an illogical or impossible situation develops and further processing is impossible. The most common cause is the destruction of the information in the stack by a user. Since the stack has read/write access, and contains dynamic variables along with link information for calls and returns, this is not uncommon. Further processing is impossible, and the operating system must abort the job. This interrupt behaves exactly like an instruction specification error, with one exception. In most cases the P register saved during the interrupt points to the instruction that caused the error. However, this error may arise when the processor is attempting to trap or exchange on another interrupt. If a trap was being attempted but could not complete, an exchange is attempted if the machine is in job mode. If the exchange is successful, the P register saved at JPS points to the instruction that originally caused the trap, and the trap exception bit is set. The operating system must abort the job at this time, but checks for the trap exception and reports the following to the user.

- The instruction executed or about to be executed when the original interrupt occurred.
- The nature of the original interrupt.
- The final reason for the job abort, which is the environment specification error.

If the machine is in monitor mode when the trap exception occurs, the machine halts, since the monitor's environment has been destroyed.

For exchange interrupts the situation is different. If an exchange from monitor to job is attempted, so that the job in question is not permitted to execute the given virtual machine to which it is exchanging, the following happens.

- The exchange from monitor to job completes, and an environment specification error is detected.
- An exchange is taken immediately from job to monitor.

The environment specification error is associated with the job and is recorded in the exchange package stored at JPS. Also, the P register saved in this exchange package is identical to the exchange package that was loaded from JPS when the original exchange from monitor to job was attempted.

If a virtual machine mismatch occurs on an attempted exchange from job to monitor, the hardware must have failed in some serious, undetected manner. The processor has no recourse other than to halt. This situation is unlikely to occur and is also difficult to detect. There is no indication in the processor error logs, and there is no indication in either the exchange package at JPS or that at MPS. An investigation of the PVA in the P register at the time of the halt (by the SMU) and an investigation of the monitor condition register (and monitor mask register), followed by a check on the registers controlling virtual machine switching, should reveal the nature of the problem. If unexplained processor halts are to be avoided, then the code in the SMU must include a check for these conditions.

External Interrupt

An external interrupt is an asynchronous interrupt. It is a signal to a processor that another processor requires some action to be performed. The identity of the processor making the request and the nature of the request must be relayed by software convention. A message buffer must be set up in central memory to contain this information. Once the request is satisfied, an exchange jump back to job continues normal processing.

Page Table Search without Find

This is a simple page fault, where a user has tried to access a page that is not in real memory. The operating system must arrange for the page to be brought into memory before processing can continue. This condition is always caught in the prevalidation of an instruction, before the point of no return. Consequently, the P register saved at JPS by the interrupt points to the instruction that could not be executed because of the missing page. In addition, the untranslatable pointer register (UTP) contains the address (PVA) that gave the page fault. To satisfy the page fault, the operating system does not have to trace back through the code being executed. It can gain all the information it requires from the UTP. Once the page fault is satisfied, an exchange back to job continues normal processing. Page faulting does not always result in loading a fresh page in memory. The operating system must apply various safeguards to ensure that a process running in a write loop does not consume all of real memory. Typically, this can be done by limiting the size of the segments being used by the user.

One last point is that certain code segments of the operating system must be wired down and not paged. This is to avoid the recurrence of faults that could otherwise occur. For example, the page fault handler itself cannot get a page fault. Such a condition normally causes a processor halt via the hierarchy mechanism of the Virtual State interrupt system.

System Call

Unlike the conditions discussed to this point, the system call condition is not an interrupt condition but a flag for the operating system monitor. The value of the corresponding bit in the monitor mask register has no effect on the setting of this flag. A process executing in job mode may need to make a request on the operating system monitor for some action. To do this, the process stores a request message in a message buffer, and then issues an exchange jump. This switches the machine state from job to monitor, and appears to monitor as if an exchange interrupt occurred. An investigation of the MCR at JPS reveals the system call flag set, and the necessary action is taken. The P register saved at JPS during the exchange points to the instruction following the exchange jump, and an exchange back to job continues normal processing. Unlike true interrupt conditions, if this flag is set by the special system instruction that modifies the MCR, an interrupt does not result.

The interrupt handler must ensure that only the MCR is checked for that condition. In the normal mechanism, the logical product of both the MM and the MCR is checked. The simplest procedure for the operating system to follow is to ensure that the MM bit (bit 10) is always set on exchange to job.

System Interval Timer

The System Interval Timer (SIT), which resides in a processor state register, is a single timer for the entire system. It is a 32-bit counter that is decremented once every microsecond. When SIT counts to zero, an interrupt is taken. This is an asynchronous interrupt and the P register stored at JPS points to the instruction following the one being executed when the SIT decremented to zero. The SIT is intended to be used for time slicing and accounting. Once the counter has decremented to zero it does not stop counting. For example, it will next decrement to -1 (2**32 -1) and continue decrementing.

Invalid Segment/Ring Number 0

The invalid segment condition bit in the MCR combines two conditions. The first of these is a true invalid segment and the second is an unlinked pointer, ring number 0. An invalid segment condition arises when either a segment descriptor table entry (SDE) has been flagged as an invalid entry (VL field = 00), or when the segment table length (STL) has been exceeded. This latter condition occurs when the segment number (SEG) portion of a PVA is greater than STL. For these conditions, the P register stored at JPS points to the instruction that attempted the central memory access that gave rise to the condition.

A ring number 0 arises when an unlinked pointer has been loaded. The operating system must arrange for the loader to form the necessary links, as described previously in the section dealing with dynamic loading. In all cases, the unlinked pointer is placed in the untranslatable pointer register (UTP), and contains all the information necessary for the operating system to form the appropriate link. When an unlinked pointer is loaded, the load completes before the interrupt is taken. The P register stored at JPS points to the instruction following the load instruction that loaded the unlinked pointer. This means the unlinked pointer was loaded into an A register, and the operating system must take care to replace this register value with the correct, linked pointer.

Outward Call/Inward Return

The ring hierarchy has been established so that procedures in inner rings can access code and data in outer rings (rings with higher numbers and lower privilege), and procedures in outer rings can call procedures in inner rings in a controlled manner. This has been described in the section on call/return. This condition has been provided to prevent a user from attempting an outward call or an inward return, and thereby causing a possible security breach. The P register stored at JPS points to the call or return instruction in question, and the operating system must either abort the job or simulate the required call. This latter process has already been described.

Soft Error Log

The soft error log bit sets when the processor encounters a hardware error that was corrected by the hardware. This includes correctable errors in the processor itself and may include, if selected, single-bit errors in central memory. Examples in the processor include a successful instruction retry, cache or MAP parity errors, and so forth. These vary from processor type to processor type. These correctable errors are also reported in the status summary register (for processor or memory), so the processor does not need to be interrupted on every correctable error. The choice may be made to ignore this interrupt (by not setting the appropriate bit in the monitor mask register), and to treat these errors in an asynchronous manner via the System Monitor Utility (SMU), a designated PP in the IOU. The P register stored at JPS points to the next instruction to be executed. If the interrupt is taken, the operating system monitor, after processing the interrupt, only has to issue an exchange to continue normal processing.

Trap Exception

Trap exception is similar to system call in that it is not an interrupt condition but is a flag to the operating system. The flag indicates that a trap interrupt was attempted, but could not be completed because a condition was encountered that prevented it. At least two other bits are set in the MCR/UCR whenever the trap exception bit is set, one being the bit that prevented the trap from completing, and the other being the bit that caused the trap. An example of this process is an arithmetic overflow encountered and a trap attempted. However, in attempting to store the SFSA, a page fault in the stack is detected and an exchange interrupt taken. After satisfying the page fault, the operating system exchanges to the user (taking care to clear the trap exception bit in the MCR), whereupon the trap takes place since the condition has not been removed from the UCR. The P register stored at JPS contains the PVA that would have been laid down in the SFSA had the trap been successful.

General Notes on the MCR

- Whenever an invalid pointer is encountered, for whatever reason, an interrupt occurs and the invalid pointer is placed in the UTP.
- Interrupts that may occur in multiples of particular interest are the four that cause entries in the UTP: invalid segment/ring number 0 (ISG), address specification error (ASE), access violation (AV), and page table search without find (PSWF). If these occur in combination, the following order of precedence applies to the PVA entered in the UTP: ISG, ASE, AV, PSWF.

USER CONDITION REGISTER (UCR)

Figure 9-3 lists the conditions recorded in the user condition register. Following are some notes on these conditions. Rather than repeat information, the arithmetic conditions have been grouped into classes that describe their behavior.

P REG	BIT NUMBER AND DEFINITION			ASSOCIATED USER MASK REGISTER BIT SET				MASK BIT CLEAR
				TRAP ENABLED		TRAP DISABLED		TRAP ENABLED OR DISABLED
				JOB MODE	MONITOR MODE	JOB MODE	MONITOR MODE	JOB OR MONITOR MODE
P	48	Privileged Instruction Fault	Mon	TRAP	TRAP	EXCH	HALT	These mask bits are permanently set.
P	49	Unimplemented Instruction	Mon	TRAP	TRAP	EXCH	HALT	
P	50	Free Flag	User	TRAP	TRAP	STACK	STACK	
P+	51	Process Interval Timer	User	TRAP	TRAP	STACK	STACK	
P	52	Inter-ring Pop	Mon	TRAP	TRAP	EXCH	HALT	
P	53	Critical Frame Flag	Mon	TRAP	TRAP	EXCH	HALT	
P+	54	Keypoint	User	TRAP	TRAP	STACK	STACK	
P	55	Divide Fault	User	TRAP	TRAP	STACK	STACK	STACK
P	56	Debug	User	TRAP	TRAP	STACK	STACK	STACK
P	57	Arithmetic Overflow	User	TRAP	TRAP	STACK	STACK	STACK
P+	58	Exponent Overflow	User	TRAP	TRAP	STACK	STACK	STACK
P+	59	Exponent Underflow	User	TRAP	TRAP	STACK	STACK	STACK
P+	60	F. P. Loss of Significance	User	TRAP	TRAP	STACK	STACK	STACK
P	61	F. P. Indefinite	User	TRAP	TRAP	STACK	STACK	STACK
P	62	Arithmetic Loss of Significance	User	TRAP	TRAP	STACK	STACK	STACK
P	63	Invalid BDP Data	User	TRAP	TRAP	STACK	STACK	STACK

Figure 9-3. User Condition Register

Privileged Instruction Fault

The privileged instruction fault is really a monitor condition and could have been implemented in the MCR. However, by recording it in the UCR, there is an opportunity for handling the fault from within the user's address space. This is the first of four monitor conditions recorded in the UCR. These conditions cannot be stacked and are termed the nonstackable conditions. In practice, the privileged instruction fault, which arises because a user has attempted to execute a privileged instruction in a nonprivileged mode, normally is handled directly by the operating system. The privileged modes of operation are fully described in volume II of the Virtual State hardware reference manual (refer to the preface for more information).

The P register stored in the SFSA points to the instruction that gave the fault. The execution of this instruction is inhibited.

Unimplemented Instruction

Unimplemented instruction is the second monitor condition flagged in the UCR. It provides a capability for emulating a model-dependent instruction with suitable software. Since the emulation should occur from within the user's address space, a trap rather than an exchange is taken. The P register stored in the SFSA points to the illegal instruction that caused the trap.

Free Flag

The free flag's purpose is to alert a process and cause the process to take some action. An example of the use of the free flag is given in the section on the call/return mechanism. In that example, an outward call is simulated by the operating system from within the user's address space. The transition from the monitor's address space to the user's address space is made by setting the free flag in the exchange package at JPS and executing an exchange jump from monitor to job. In the prevalidation of the next instruction to be executed in the user's job, the free flag is detected and a trap taken. The P register stored in the SFSA points to the next instruction to be executed in the user's code. This UCR condition is unique in that it takes priority over MCR conditions that may arise at the same time.

Process Interval Timer

The Process Interval Timer (PIT) is a 32-bit counter that decrements once every microsecond. Each process has a unique counter when it is in execution. Whenever a PIT reaches 0, a condition bit sets in the UCR, and if it is enabled, a trap is taken. The PIT continues counting at this time. The counter assumes a value of -1 (2**32-1) and the decrementing continues 1 μs after the PIT has zeroed. This condition is an asynchronous interrupt, similar to the SIT that is recorded in the MCR. The P register saved in the SFSA points to the next instruction to be executed. In other words, when the trap handler has completed its processing and issued a return, normal instruction execution resumes.

Inter-ring Pop

The inter-ring pop is the third monitor condition recorded in the UCR. The pop instruction is described in the section on the call/return mechanism. Its function is to dispose of stack frames, typically during cleanup when a process is being terminated. The pop instruction merely moves pointers (DSP, CSF, PSA, and TOS) that point at a given stack. It does not contain any of the safeguards required when crossing rings. If a ring crossing is attempted in the cleanup process, a trap is taken and software procedures are called to ensure that the ring crossing takes place in a controlled manner. The P register saved in the SFSA points to the pop instruction that attempted the ring crossing, and whose execution was inhibited.

Critical Frame Flag

The critical frame flag (CFF) is the fourth and final monitor condition recorded in the UCR. The CFF is a software flag acted on by the hardware. Software sets this flag to prevent the disposal of certain stack frames that may be shared by separate tasks running in the same address space. The flag is cleared by a call and trap, so that each instance of a procedure begins in a noncritical state. It is likewise restored on a pop or a return for the criticality of the current stack frame to be determined. This condition provides an interrupt into the user's address space, and the trap handler must determine how the stack frame can be disposed of. In other words, the criticality of the stack frame is set by software convention, and any alteration of the criticality or disposal of the stack frame must be under the control of the same software. The P register, saved in the SFSA, points to the return or pop instruction that attempted to eliminate the critical stack frame.

Keypoint

The keypoint condition indicates that software is to collect hardware performance data at this point of the program. A full discussion of this topic is in volume II of the Virtual State hardware reference manual (refer to the preface for more information). In this case, the P register saved in the SFSA points to the instruction following the keypoint instruction that caused the trap.

General Notes on the UCR

- The first seven conditions recorded in the UCR have just been described. They comprise four monitor conditions and three user conditions. The bits in the user mask (UM) register corresponding to these seven conditions are permanently selected by the hardware. If one of these conditions arises and traps are enabled, a trap is taken.

- For the four nonstackable monitor conditions, execution of the instruction causing the trap is always inhibited. Furthermore, the offending instruction is rarely if ever executed. However, since the P register saved in the SFSA points to the offending instruction, the trap handler must advance the value of the P register saved in the SFSA before issuing a return.

Debug

A debug indicates that a condition, such as a storage reference made or branch taken, was met. For a full discussion of the facilities in this area, refer to the section on debug. The P register saved in the SFSA points to the instruction that caused the trap to occur.

Invalid BDP Data

Invalid BDP data indicates that a BDP instruction encountered data not matching the required format. In this case, the P register saved in the SFSA points to the instruction that encountered the invalid BDP data.

Arithmetic Conditions

The remaining seven user conditions are arithmetic conditions. These are:

- Divide fault.
- Arithmetic overflow.
- Floating-point indefinite.
- Arithmetic loss of significance.
- Exponent overflow.
- Exponent underflow.
- Floating-point loss of significance.

These conditions fall into one of two classes as follows:

- The P register stored in the SFSA points to the instruction that caused the fault.
- The P register points to the instruction following the one that caused the fault.

In addition, the instruction may or may not be executed before the trap is taken.

The general intent of Virtual State is to be able to identify the instruction that caused the fault. This means that the P register saved in the SFSA normally points to the instruction in question. This is particularly important, since it is impossible to back up the instruction stream when an instruction has executed and the P register has been advanced. (It follows automatically that the instruction execution is normally inhibited.)

Conditions Where the Instruction Is Inhibited

For this class of arithmetic faults, the execution of the instruction that caused the fault is inhibited, and the P register saved in the SFSA points to that instruction. Exceptions are noted in the following text. The conditions that fall into this category are:

- Divide fault (integer, decimal, floating-point).
- Arithmetic overflow (integer, decimal).
- Floating-point indefinite.
- Arithmetic loss of significance (integer, decimal).

General notes:

- Floating-point indefinite falls into this category because it can arise on a branch instruction (32 bits) as well as on an arithmetic operation (16 bits). Therefore, it is not possible to back up the instruction stream.
- A divide fault occurs either on a divide by zero, or when the divisor is an unnormalized floating-point number. The latter case does not necessarily result in a divide fault, but the single and double precision quotient operations do not prenormalize. (However, all floating-point operations postnormalize to the extent that normalized numbers emerge if normalized numbers are input to the floating-point unit.) Also, if traps are disabled or the divide fault interrupt is not selected, the instruction is still inhibited and execution continues at the next instruction in sequence.
- Traps on user conditions have been included for convenience. They may be selected or disabled by the user via the UM, and cause an interruption to the normal execution sequence. When the condition has not been selected but is encountered, the appropriate bit is still set in the UCR, and may be tested and cleared by a special instruction, called the branch on condition register.

Conditions Where the Instruction Is Executed

For this class of arithmetic faults, the execution of the instruction that caused the fault is completed, and the P register saved in the SFSA points to the next instruction. The conditions that fall into this category are:

- Exponent overflow.
- Exponent underflow.
- Floating-point loss of significance.

It is important that the instructions that occurred while the condition arose are executed, since the Virtual State floating-point format has been chosen so that a true result is returned, even though an exponent overflow or underflow occurs. It provides a programmer with the opportunity to scale variables and continue processing if desired. This is fully described in volume II of the Virtual State hardware reference manual (refer to the preface for more information).

Vector Instructions

The vector instructions require some special attention because of the multiple operands involved. Floating-point vectors may encounter up to four of these conditions. The vector instruction execution is not inhibited, and the interrupt occurs after the completion of the vector instruction.

If a single condition is encountered, the P register saved in the SFSA follows the pattern for scalar instructions.

- P points to the following instruction for:

  Exponent overflow

  Exponent underflow

  Floating-point lost of significance

- P points to the vector instruction that encountered the interrupt for:

  Divide fault

  Arithmetic overflow

  Floating-point indefinite

  Arithmetic loss of significance

If multiple conditions are encountered requiring different values of P, the P always is set to point to the instruction following the vector instruction that encountered the multiple conditions.

SIMULATED INTERRUPTS

There are two ways in which an interrupt can be artificially generated. A bit can be set in the UCR (by a branch on condition register instruction), and a bit can be set in the UM when the corresponding bit is already set in the UCR. In both instances, the P register saved in the SFSA points to the instruction following the instruction that set the bit in either a mask register or a condition register, that is, following a branch on condition register (BCR) or a copy. It is not good practice to set bits in a UCR. This facility has been included as a diagnostic aid, to verify that the interrupt system is functioning correctly.

MULTIPLE INTERRUPTS

When more than one interrupt condition arises at one time the following rules apply.

- Exchange interrupts are always serviced by the hardware before trap interrupts.
- Multiple interrupts of the same type are all recorded in the condition registers, when the interrupts occur simultaneously. The precise mechanism is processor-model dependent, with some processors recording all coincident conditions simultaneously, and others recording them one at a time. The interrupt handlers must accommodate multiple, simultaneous interrupts.

Multiple interrupts arise because of the asynchronous nature of certain interrupt conditions. Since the P register saved by the interrupt is intended to point to either the instruction that caused the fault or to the following instruction, it is important to understand what is contained in that register. The following general rules should help.

- A detected uncorrectable error (DUE) always takes precedence and leaves the P register in an undefined state, unless PND is set.
- The synchronous interrupts take priority over the asynchronous interrupts.

Care must be taken when designing and generating interrupt handlers, and these rules must be applied at all times. The following example clarifies the pitfalls.

When an asynchronous interrupt occurs (for example, a PIT), the P register saved in the SFSA points to the next instruction to be executed. The trap is taken and processed and a return is issued, which continues normal processing. However, if a PIT occurs simultaneously with, for example, an unimplemented instruction, the P register saved in the SFSA points to the unimplemented instruction. If only the PIT is acted on, the return causes the unimplemented instruction fault to be detected again. However, if only the unimplemented instruction is acted on, the PIT is never seen. This occurs because on a trap the UCR is saved in the SFSA and the live UCR is zeroed. It is the live UCR that carries back across the return. This is different from the exchange mechanism, where a fresh copy of the condition registers is invoked (either from the exchange package at MPS or that at JPS) for each exchange interval. If an exchange condition is processed, but the bit in the condition register in the exchange package in memory is not cleared, an exchange loop follows.

Probably the safest rule to follow is to process all conditions that have arisen, and that have been selected, at one time.

Finally, software cooperation is needed when interrupts are caused artificially (for example, by setting the UM dynamically). If the normal action taken by the interrupt handler is to advance the P counter, an instruction is omitted in this case. For example, in the sequence:

```
ENTE    XF,X'E6'
ENTP    XE, 1
CPYXS   XE, XF
LX      XF, A5, ABC
```

the LX instruction may be spaced over by the interrupt handler. To avoid such an occurrence, it is recommended that a do-nothing (CPYXX X0,X0) be inserted immediately following the instruction causing the trap. In this case the instruction is (CPYXS XE,XF).

Virtual State processors provide a debug facility to assist programmers debugging at the machine code level in Virtual State. The operation of debug is fairly complex since, in effect, it provides an interrupt capability during an instruction execution. In practice, the instruction is not executed until the debug processing is complete, although prevalidation of the instruction may complete prior to the debug. As a result of this flexibility, the state of the process must be retained across interrupts, and several process state registers have been defined for this purpose.

The user may elect to debug based on a number of conditions. These are:

- Whenever data is read from a specified area in virtual memory.
- Whenever data is written into a specified area in virtual memory.
- Whenever an instruction is fetched from a specified area in virtual memory.
- Whenever a branch is made to a specified area in virtual memory.
- Whenever either a call indirect or a call relative instruction is issued to a procedure in a specified area in virtual memory.

For any instruction issued, the user may elect to debug on any combination of these conditions, for up to 32 different areas in virtual memory. The conditions are specified in a debug list (figure 10-1) provided by the user, and further controlled by the debug mask (DM) register. Finally, debugging is activated by setting bit 56 in the user mask (UM) register and enabling traps.

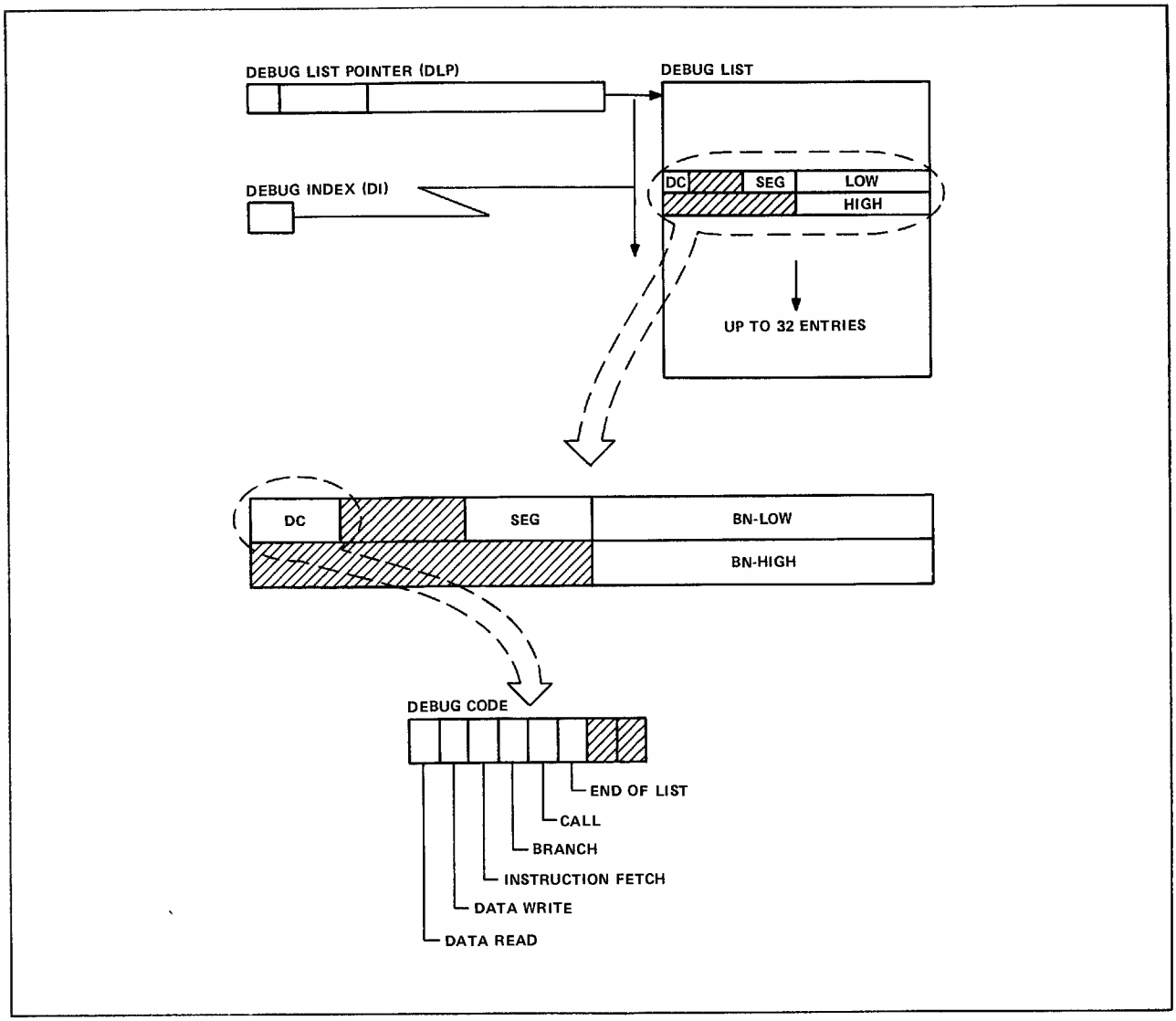


Figure 10-1. Debug List

Each entry in the debug list consists of two words, which must be placed on word boundaries. The two words describe the debug conditions, the segment in virtual memory to which the conditions apply, and the range of addresses (byte numbers) in the segment to which the debug conditions are restricted (figure 10-2).

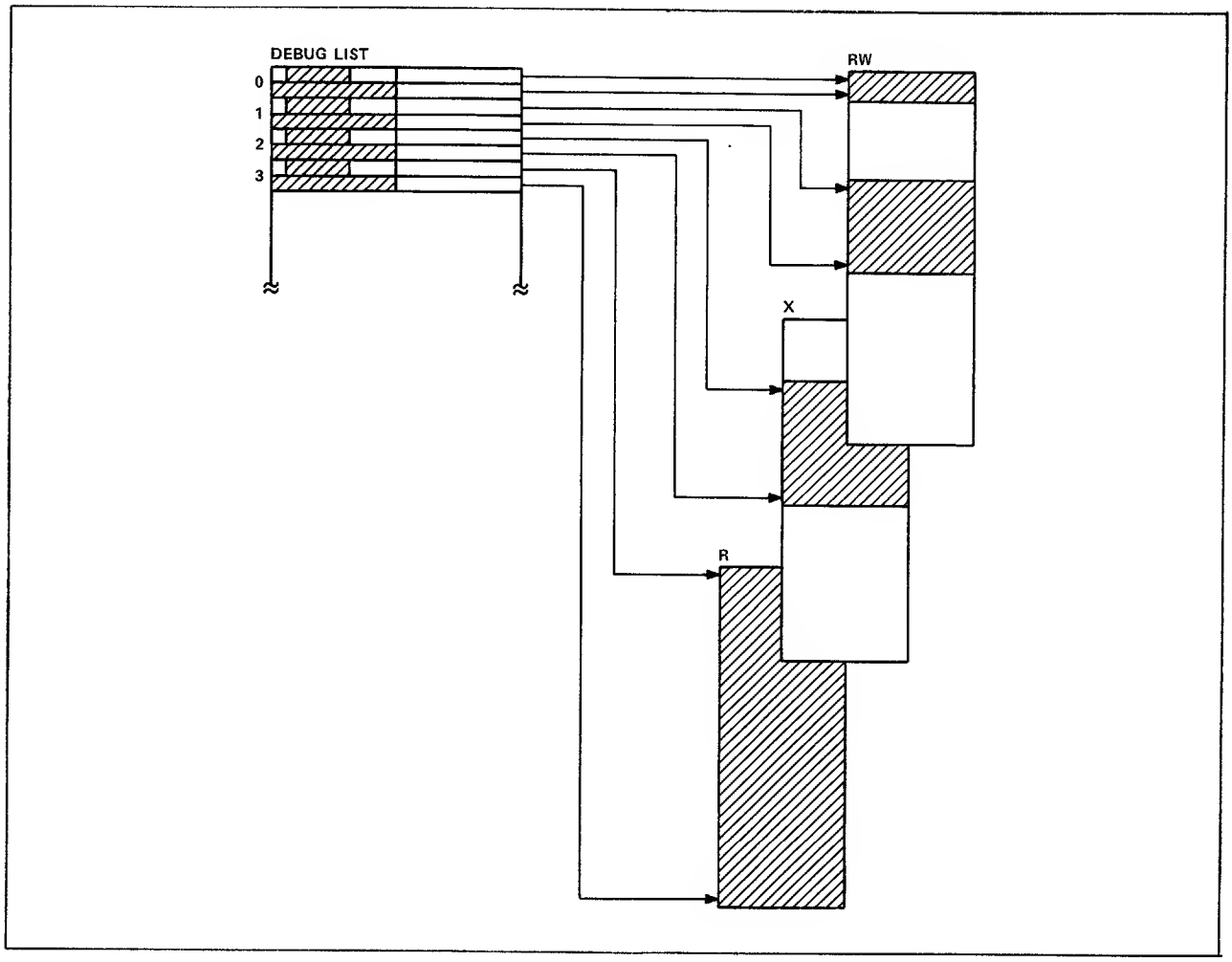


Figure 10-2. Debug List Entries

The debug code (DC) can be selectively activated at run time by setting the DM appropriately (figure 10-3). For each condition set in the DC there is a corresponding condition select in the DM. Debugging occurs only when there is a coincidence between a condition bit in the DC and a condition select bit in the DM.

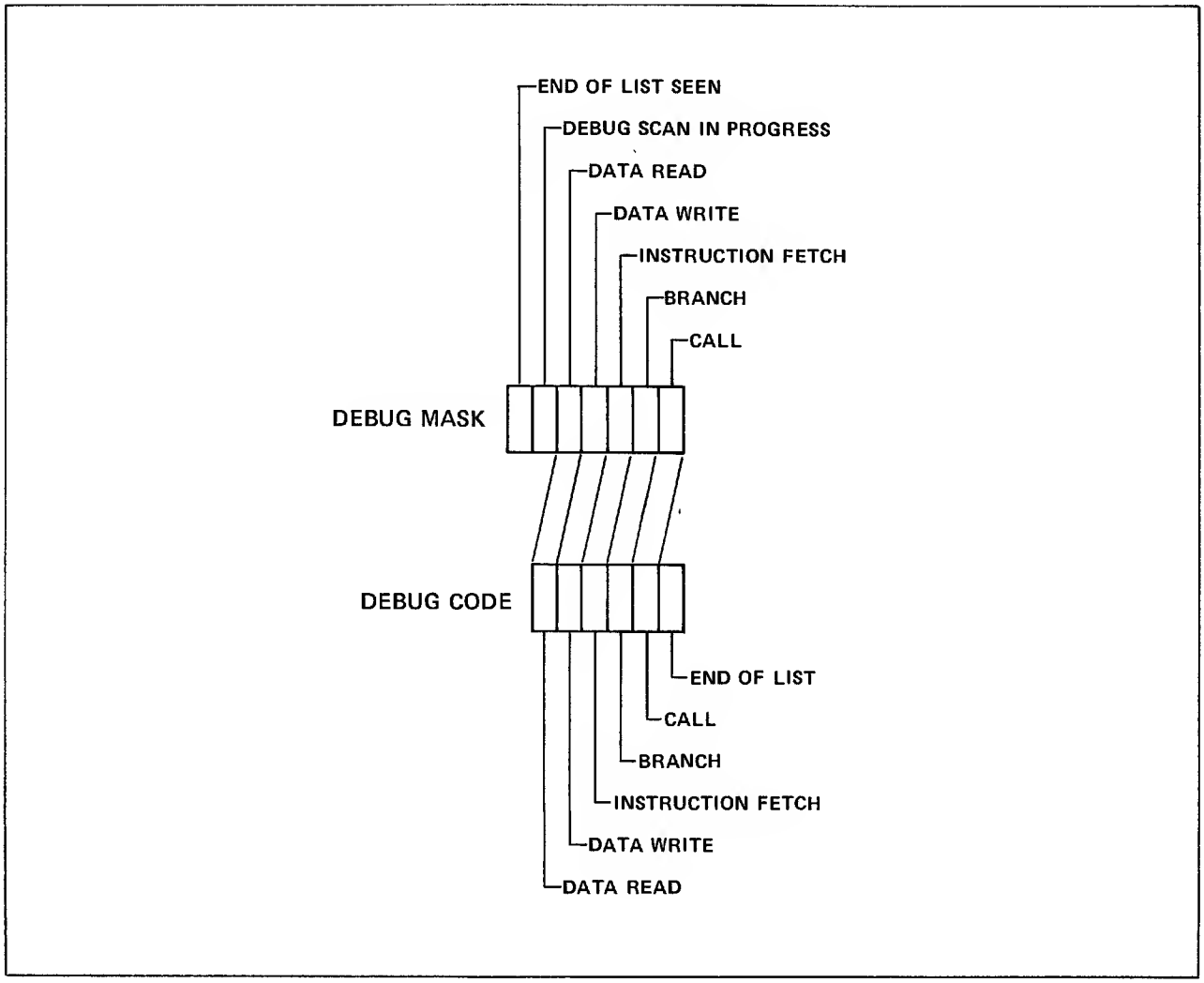


Figure 10-3. Debug Condition Select

A user may insert a debug list into his program, set the DM to select a range of conditions, but run in a normal mode by not choosing to trap on debug. In this case, the overhead due to debugging is zero. Performance degradation due to debug testing occurs whenever the first two items in the following list are true. However, a debug trap does not occur unless all of the following are true.

- Traps are enabled.
- Bit 56 in the UM is set (debugging selected).
- The process is Virtual State.
- The DM and DC registers both select a test that is satisfied for the current instruction.
- The end of list seen flag in the DM is not set.

When these conditions are met the debug list is scanned and trap interrupts taken, as required, by the hardware setting bit 56 in the UCR.

The hardware uses three process state registers to control scanning of the debug list. These are the debug list pointer (DLP), the debug index (DI), and the DM. The DLP gives the starting address of the debug list; the DI keeps track of the position within that list; and the DM contains two flags to control the initiation and termination of the debug. The first of these flags is the debug scan in progress flag, which controls the start of the process. Conceptually, whenever an instruction is executed, this flag is cleared. Then when the next instruction is issued and debug is active, the processor starts scanning the debug list from the beginning. The second flag is the end of list seen flag, which controls the termination of debugging for the current instruction. This flag is set when either 32 entries in the debug list have been scanned, or when an end of list bit is encountered in a debug code. Again, conceptually the flag is cleared whenever an instruction is executed. The complete, conceptual hardware process is shown in figure 10-4. These flags are primarily hardware flags that have been included in a process state register so the hardware can remember where it is when an interrupt is taken. If they are set by software, they could perturb the operation of debug.

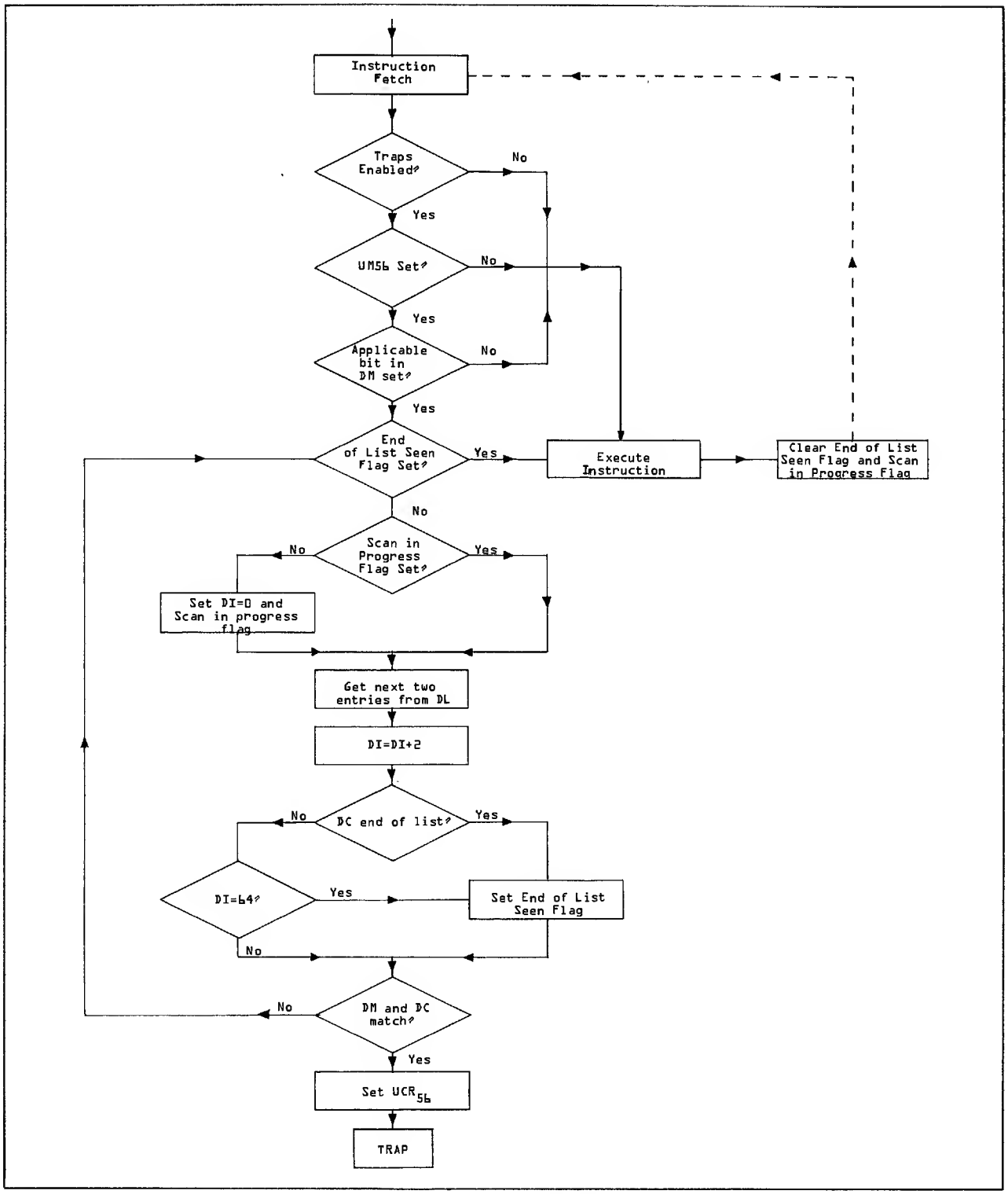


Figure 10-4. Conceptual Debug Procedure

General notes on debug:

- Debugging only occurs when traps are enabled. The interrupt handlers cannot make use of this facility. However, monitor mode code can make use of debug, providing traps are enabled.

- Debug list entries beyond the 32nd are ignored regardless of whether or not an end of list seen flag has been encountered in the DC.

- The user must have been granted local privilege in order to alter the DLP. In other words, local privilege is required to specify a different debug list in the same process.

- Several instructions apply to more than one debug condition. For example, many BDP instructions can trap on both read and write since they have both source and destination operands in memory. One instruction, call indirect, applies to four debug conditions (it is a call, it reads from the binding section, it writes into the SFSA, and it is fetched).

- Also, some instructions have more than one operand checked for a debug trap. Typically, these are instructions that specify the address of a table to be used in conjunction with two operands (for example, translate and edit).

- Exchange jumps, which branch to a value found in an exchange package at a real memory address, do not cause a debug trap on branch. Similarly, compare and swap does not cause a debug trap on branch when it rejects as a result of a hardware lock set.

- When a debug trap occurs the P register stored in the SFSA points to the instruction that would have been executed had the debug trap not been taken. Also, when a debug trap is taken, the DI has an odd value. That is, it points to the second word of a word pair entry in the debug list. However, while the debug list is being scanned, interrupts are enabled, and should an asynchronous interrupt occur, such as PIT or external interrupt, the DI may be either odd or even.

This section will be supplied later.

A Register

Address register.

AO/R

Architectural objectives/requirements.

ASE

Address specification error.

ASID

Active segment identifier.

AV

Access violation.

BC

Base constant.

BCO

Branch on condition.

BCR

Branch on condition register.

BCT

Between command test.

BDP

Business data processing.

BN

Byte number.

BS

Binding section.

CBP

Code base pointer.

CCD

Charge-coupled devices.

CEL

Corrected log error.

CEM

Configuration environment monitor.

CF

Critical frame.

CFF

Critical frame flag.

CM

Central memory.

CMA

Central memory access.

CMU

Compare-move unit.

CPU

Central processing unit.

CRT

Cathode ray tube.

CSF

Current stack frame.

DAP

Design action paper.

DC

Debug code.

DEC

Model-dependent environment control (compare EC).

DI

Debug index.

DLP

Debug list pointer.

DM

Debug mask.

DMR

Debug mask register.

DSP

Dynamic space pointer.

DUE

Detected uncorrectable error.

EBAM

Electron beam accessed memory.

EC

Environment control.

ECC

Error correction code.

ECL

Emitter-coupled logic.

ECM

Extended central memory.

ECS

Extended core storage.

EDMS

European data management system.

EID

Element identifier.

EM

Exit mode.

EPF

External procedure flag.

ES

End suppression toggle.

FL

Field length.

FLC

Central memory field length register.

FLE

Extended core storage field length register.

FTN

FORTRAN.

ILH

Instruction look-ahead.

I/O

Input/output.

IOU

Input/output unit.

ISG

Invalid segment/ring number 0.

JEP

Job mode exchange package.

JPS

Job process state.

KC

Keypoint code.

KCN

Keypoint class number.

KEF

Keypoint enable flag.

KM

Keypoint mask.

LED

Light emitting diode.

LPID

Last processor identification.

LRU

Least recently used.

LSI

Large-scale integration.

MA

Monitor address.

MAC

Maintenance access control.

MCH

Maintenance channel.

MCI

Maintenance channel interface.

MCR

Monitor condition register.

MDF

Model-dependent flags.

MDW

Model-dependent word.

MEP

Monitor mode exchange package.

MF

Monitor flag.

MID

Maintenance identifier.

MIGDS

Model-independent general design specification.

MM

Monitor mask.

MOP

Micro-operator.

MOS

Metal-oxide semiconductor.

MPS

Monitor process state pointer.

MTR

Monitor.

MUX

Multiplexer.

NS

Negative sign toggle.

OCF

On-condition flag.

OI

Options installed.

ON

Occurrence number.

OP

Operation code.

P Register

Program address register.

PCO

Printed circuit operation.

PFA

Page frame address.

PFS

Processor fault status.

PID

Processor identifier.

PIT

Process interval timer.

PMF

Performance monitoring flag.

PN

Page number.

PND

Processor not damaged.

PO

Page offset.

PP

Peripheral processor.

PPM

Peripheral processor memory.

PPS

Peripheral processor subsystem.

PPU

Peripheral processor unit.

PSA

Previous save area.

PSF

Previous stack frame.

PSM

Page size mask.

PSR

Process state registers.

PSWF

Page table search without find.

PTA

Page table address.

PTE

Page table entry.

PTL

Page table length.

PTM

Processor test mode.

PVA

Process virtual address.

RA

Reference address.

RA/FL

Reference address/field length.

RAC

Central memory reference address register.

RAE

Extended core storage reference address register.

RAM

Random access memory.

RMA

Real memory address.

RN

Ring number.

RNI

Read next instruction.

ROM

Read-only memory.

RP

Read access control (segment descriptor field).

SCL

System command language.

SDE

Segment descriptor table entries.

SDT

Segment descriptor table.

SECDED

Single-error correction/double-error detection.

SEG

Process segment number.

SFSA

Stack frame save area.

SIT

System interval timer.

SMU

System monitor utility.

SPID

Segment page identifier.

SPT

System page table.

SRT

Subscript range table.

SS

Status summary.

SSP

Subsystem procedure.

STA

Segment table address.

STL

Segment table length.

SVA

System virtual address.

TE

Trap enable.

TED

Trap enable delay.

TEF

Trap enable flip-flop.

TOS

Top of stack.

TP

Trap pointer.

UCR

User condition register.

UM

Used/modified control (page descriptor table).

UM

User mask.

UTP

Untranslatable pointer.

UVMID

Untranslatable virtual machine identifier.

VC

Search control (page descriptor field).

VL

Segment validation (segment descriptor field).

VMCL

Virtual machine capability list.

VMID

Virtual machine identifier.

WP

Write access control (segment descriptor field).

XP

Execute access control (segment descriptor field).

INDEX

COMMENT SHEET

MANUAL TITLE: CDC CYBER 170 Models 815, 825, 835, 845, and 855, CDC CYBER 180 Models 810, 830, 835, 840, 845, 850, 855, 860, and 990 Computer Systems, and CDC CYBER 845S, 855S, 840A, 850A, 860A, 990E, and 995E Computer Systems General Description HMM

PUBLICATION NO.: 60459960 **REVISION:** E

NAME: _____

COMPANY: _____

STREET ADDRESS: _____

CITY: _____ **STATE:** _____ **ZIP CODE:** _____

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments below (please include page number references).

☐ Please Reply ☐ No Reply Necessary

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND TAPE

Comments (continued from other side)

ease fold on dotted line;
al edges with tape only.

FOLD

OLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

rst-Class Mail Permit No. 8241 Minneapolis, MN

POSTAGE WILL BE PAID BY ADDRESSEE

NTROL DATA
nical Publications
219
N. Lexington Avenue
Hills, MN 55126-9983

CORPORATE HEADQUARTERS, P.O. BOX O, MINNEAPOLIS, MINN 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S

CONTROL DATA